Approximate Caching for Efficiently Serving Text-to-Image Diffusion Models

Shubham Agarwal and Subrata Mitra, *Adobe Research;* Sarthak Chakraborty, *UIUC;* Srikrishna Karanam, Koyel Mukherjee, and Shiv Kumar Saini, *Adobe Research*

https://www.usenix.org/conference/nsdi24/presentation/agarwal-shubham

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Approximate Caching for Efficiently Serving Text-to-Image Diffusion Models

Shubham Agarwal
Adobe Research

Subrata Mitra[*]
Adobe Research

Sarthak Chakraborty[†]
UIUC

Srikrishna Karanam
Adobe Research

Koyel Mukherjee
Adobe Research

Shiv Kumar Saini
Adobe Research

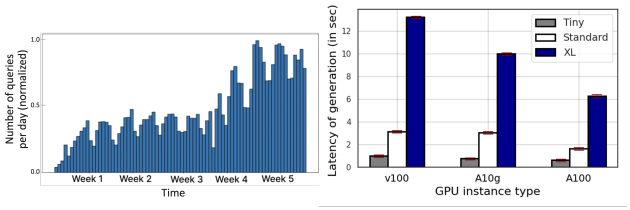
Abstract

Text-to-image generation using diffusion models has seen explosive popularity owing to their ability in producing high quality images adhering to text prompts. However, diffusion-models go through a large number of iterative denoising steps, and are resource-intensive, requiring expensive GPUs and incurring considerable latency. In this paper, we introduce a novel *approximate-caching* technique that can reduce such iterative denoising steps by reusing intermediate noise states created during a prior image generation. Based on this idea, we present an end-to-end text-to-image generation system, NIRVANA, that uses approximate-caching with a novel cache management policy to provide 21% GPU compute savings, 19.8% end-to-end latency reduction, and 19% dollar savings on two real production workloads. We further present an extensive characterization of real production text-to-image prompts from the perspective of caching, popularity and reuse of intermediate states in a large production environment.

1 Introduction

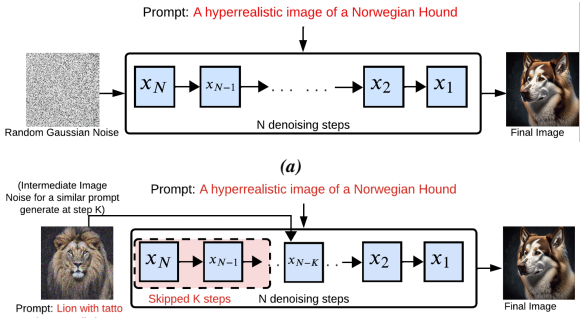
Text-to-image generation has drastically matured over the years [34, 87] and has now become a widely popular feature offered by various companies [8, 14], being integrated into various new creative workflows [10]. The popularity of text-to-image models has become massive. Adobe recently reported [10] that over 2 billion images were created using Firefly [9] text-to-image service. Similar popularity has also been reported for Dall-E-2 from OpenAI [12]. Figure 1a shows the staggering growth over time in the numbers of prompts submitted to a portal running stable-diffusion-based text-to-image model, as captured by the DiffusionDB dataset [79].

State-of-the-art in text-to-image: In text-to-image generation, given a text *prompt* describing certain desirable characteristics of an image, a deep neural network model generates an image capturing the descriptions provided in the prompt. While researchers have been attempting to design viable and consistent text-to-image models for quite some time using VAEs [52], GANs [42] and other techniques [37], the most popular text-to-image production systems of today [9, 17] are based on *Diffusion-Models* (DMs) [36, 49]. The widespread adaptation of DMs can be attributed to their capability to generate superior quality images and to condition the generation more accurately according to the input prompt.



(a) *(b)*

Figure 1: (a) Shows normalized growth in workload over 5 weeks in DiffusionDB. (b) Latency for `tiny`, `standard` and `XL` stable-diffusion models from Hugging Face [51] repository on three different GPU architectures in AWS. The image quality produced by the larger `XL` model is far superior to the smaller two models but comes with a substantially large latency overhead which also implies higher costs.



(a)

(b)

Figure 2: (a) Shows how vanilla `DMs` work. (b) Shows how `DMs` with approximate-caching works, where first K denoising steps are skipped after an intermediate noise belonging to a different prompt present in the cache is retrieved and reused.

Problems with `DMs`: Text-to-image products should offer real-time interaction like search platforms. However, image generation using DMs is a computational resource-intensive task and suffers from relatively long latency. During the inference phase in DMs (that is generating an image from text), essentially a Gaussian noise is iteratively denoised, using the

[*]Corresponding author (subrata.mitra@adobe.com)
[†]Work done at Adobe Research

input prompt as a condition, to produce the output image in a Markov process [39]. The traditional DMs use as much as 1000 such iterative diffusion steps [36] for this. Some subsequent optimizations [61,76] enable such denoising to be done with approximately 50-100 iterative steps [2]. Still, even with 50 iterative steps, image generation with DMs is a resource-intensive and slow task that prohibits interactive experience and results in huge computational costs on expensive GPUs.

In Figure 1b, we show the latency (in seconds) for different GPU architectures (i.e., A100, A10g, and V100 from NVIDIA) that are available on Amazon Web Services (AWS) and for a few DMs of different sizes (i.e., # of parameters): `Stable Diffusion XL`, `Stable Diffusion 2.1`, `Tiny Stable Diffusion` from Hugging Face repository [51] with typical 50 diffusion steps for image generation during inference. It can be observed that while smaller models (e.g., `Tiny` and `Standard`) can provide significantly low inference latency compared to a larger model(e.g., `XL`), it usually comes with significant degradation in the quality of the generated image [67]. Therefore, using a smaller model to reduce latency for better user experience might actually defeat the purpose. Latency can also be reduced by using a more powerful GPU for inference, as can be observed in Figure 1b, but pricing for cloud instances with powerful GPUs such as NVIDIA A100s is significantly high. For instance, in the US East region [1] the V100 is priced at approximately $3.06 per hour, the A10g at $8.144 per hour, and the A100 at $32.77 per hour making an A100 instance 4X costlier than A10g and more than 10X costlier than a V100. Therefore, while inference latency can be reduced with more powerful GPUs, inference cost per image also significantly increases.

In this paper, we introduce an efficient text-to-image generation system called NIRVANA that uses novel *approximate caching* technique to significantly reduce the computational cost and latency by effectively reusing intermediate states created during image generation for prior prompts.

Figure 2 illustrates the key idea behind NIRVANA. For an input prompt shown in *red* in Figure 2a we show how the vanilla or standard DMs work through multiple iterative denoising steps. Here, the process starts from a Gaussian noise at state x_N and then performs N denoising steps with the input prompt as the condition to finally produce a coherent image in state x_0. Figure 2b shows how in *approximate caching*, first K steps are skipped, and directly a suitable intermediate noise from a different prompt is retrieved and used. The value of K depends on the similarity between the new input prompt and the prompt from which the noise was retrieved. Therefore, the amount of compute and latency savings can vary across prompts based on the availability of similar prompts in the cache that stores the intermediate noises/states produced during the image generation from previously encountered prompts. The phrase *approximate caching* emphasizes the fact that in this system, we are not directly reusing the retrieved object from the cache, rather we are retrieving an

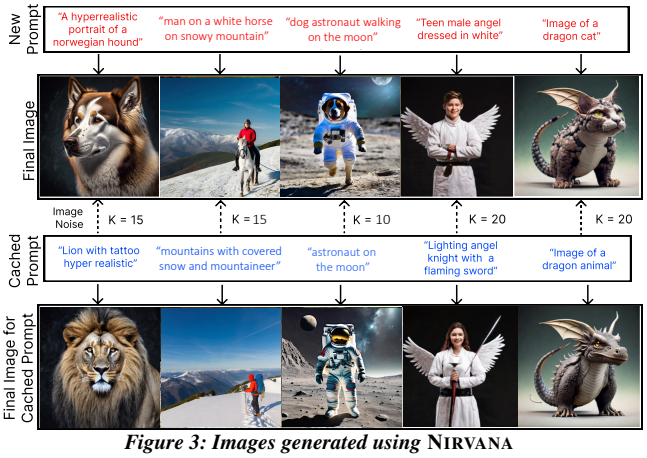


***Figure 3: Images generated using* NIRVANA**

intermediate state and further conditioning those to tailor the generated image according to the new prompt. Thus, NIRVANA is very different from a retrieval-based system such as GPT-CACHE [7], PINECONE [4] that proposes direct retrieval of an image from a cache based on the input prompt.

NIRVANA selects the value of K for a particular input prompt by effectively controlling the *hit-rate* vs. compute savings trade-off. *Hit-rate* here means how likely an incoming prompt can be matched to a *similar enough* prompt in the cache, while K denotes the # steps we skip at the beginning of the DMs when using the retrieved noise from the cache. The *hit-rate*, can be made significantly high if we are planning to skip a very small number of steps at the beginning (i.e., lower value for K). The reason being, if we are skipping less diffusion steps at the beginning, noise from very dissimilar prompts can still be denoised effectively to produce a coherent image. While for high K, the scope for further denoising conditioned on the new prompt becomes limited. However, it provides huge compute savings when done for certain amenable prompts. Careful design of NIRVANA can navigate this complexity by calculating a suitable value for K for each incoming prompt to maintain high quality as well as optimize for maximum compute savings.

Figure 3 shows some real examples using prompts from DiffusionDB [79] to illustrate how NIRVANA can transform a noise from a seemingly different prior prompt to a coherent and high-quality image while providing significant latency and compute reduction at the same time.

Furthermore, in NIRVANA, we design a novel approximate-cache management policy, called *Least Computationally Beneficial and Frequently Used* (`LCBFU`), that manages the storage of noises in such a manner that for a given cache storage size, we optimize the space for the noises that is likely to give the best computational efficiency to NIRVANA.

Overall, NIRVANA can maintain generated image quality very close to vanilla diffusion-models (i.e., DMs without approximate-caching) while providing 21 % savings in GPU computation, 19.8 % reduction in latency, and 19% amortized savings on dollars spent on image generation. Our study in-

volving 60 users with 1000 images shows that 79% users liked NIRVANA generated images which is far better than the best baseline GPT-CACHE [7] with only 31% likes, and much closer to the quality of images generated by the expensive and slow VANILLA model, liked by 86% users.

We summarize our contributions in this paper as follows:

1. We introduce the novel idea of *approximate caching* that provides significant computation saving in the production pipeline of diffusion models for text-to-image generation.
2. We propose an effective cache-management mechanism, called LCBFU, that can optimize the reuse of computation states and computation savings.
3. We present end-to-end design details and rationale for NIRVANA, which is our optimized text-to-image deployment system on the cloud.
4. We characterize real production prompts for text-to-image models from the perspective of reusability and caching.
5. We present extensive evaluation with two real and large production prompts from text-to-image models, along with a human evaluation and several sensitivity studies.

2 Background

2.1 Diffusion Models (DMs)

Diffusion models (DMs) progressively denoise a random Gaussian noise to generate an image conditioned on text. The training procedure contains a forward diffusion process, which obscures an image by adding noise repeatedly in a Markov process until it saturates to Gaussian noise. In the backward diffusion process, the original image is recovered by removing noise repeatedly. Each denoising step is called "sampling" since the model generates a sample by removing noise, and the method used for sampling is called the *sampler*.

In the forward diffusion process, Gaussian noise gets progressively added to an initial image x_0 for T steps to get x_T. With the Markov chain assumption, it is expressed as:

$$q(x_{1:T}|x_0) := \prod_{t=0}^{T} q(x_t|x_{t-1}) \tag{1}$$

$$q(x_t|x_{t-1}) := \mathcal{N}(x_t|\sqrt{1-\beta_t}x_{t-1}, \beta_t I) \tag{2}$$

where $q(x_t|x_{t-1})$ is the posterior probability, and $\beta_1, ..., \beta_T$ is the noise schedule (either learned or fixed) to regulate the noise level at each diffusion step. Similarly, the backward diffusion process can be written as:

$$p_\theta(x_{0:T}) := p(x_T) \prod_{t=0}^{T} p_\theta(x_{t-1}|x_t) \tag{3}$$

$$p_\theta(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}|\bar{\mu}_t, \bar{\beta}_t I) \tag{4}$$

where $p_\theta(.)$ denotes the probability of observing x_{t-1} given x_t. Here, $p(x_T) = \mathcal{N}(x_T|0, I)$. Here, $\bar{\mu}_t$ and $\bar{\beta}_t$ is learned. The objective is to learn $p(\theta)$ that maximizes the likelihood of training data in the backward/reverse diffusion process. Recent optimizations approximate the backward diffusion process by skipping certain intermediate states at predetermined timesteps [54, 90], thus reducing inference steps from $T \approx 1000$ steps to $N \approx 50$ steps. This is achieved by learning a sampler that predicts how much noise will remain after T/N step for every one diffusion step. Notably, each step consumes equal time and compute as it uses the same denoising process on the same diffusion model [87]. Even with these optimizations, image generation still takes 10 seconds on A10g and 6 seconds on A100 GPUs for large models (Figure 1b).

Diffusion model backbone is based on the U-Net architecture [70]. When a text prompt is given, the image generation process is conditioned through cross-attention within the model [69]. Thus, it develops a text-to-image framework capable of generating visually coherent and contextually relevant images based on textual descriptions.

In practice, each x_i can either be the actual image or its latent representation computed by an image encoder [69]. Generally, the latter approach, termed as latent diffusion model (LDM) is preferred since it captures the hidden characteristics of an image. We use the same in our work. To generate the final image from its latent space x_0 at the end of backward diffusion, x_0 is decoded with the inverse of the image encoder.

2.2 Dynamics of Image Generation

The amount of reconditioning needed for the retrieved noise to suit the new prompt depends on K. We observe that various concepts/characteristics of an image, like, *layout*, *color*, *shape or objects*, *style*, etc. are not easy to modify beyond certain K. With initial noise being random Gaussian, similar to recent works [89] (see **Appendix** E for details), we observed for our dataset that LDM models [69] decide the layout of the objects first within the initial ∼20% of diffusion steps. Color map for the overall image then gets decided within ∼40% steps followed by the shape and the size of the objects and then the artistic style of the image. Overall, we observed that after ∼50% of the steps, the concepts get frozen and no further attempts to recondition the intermediate image are reflected in the final generated image. For example, when attempting to recondition the image of a brown horse grazing in a green field (Figure 4), the color can be modified after $K = 10$, but not after $K = 15$. Similarly, actions, objects, and backgrounds could be modified only till certain steps and not beyond that.

3 Understanding User Prompts

We first show some characterization of real user prompts in DiffusionDB dataset [79] having 2 million prompts ranging from 8 Aug, 2022 to 22 Aug, 2022. Each entry has the user-id, prompt, timestamp, and the generated image in the dataset.

(1) Top-Y% most popular prompt clusters over days

We cluster the prompt queries for each day using DBSCAN algorithm [73] on their CLIP text embeddings [68] with multiple *eps* values (*eps* controls the maximum distance between two samples for one to be considered the neighbor of other). We find top-Y% (with Y = 1, 2, 5, and 10) clusters and extract the most popular prompts on Day 1. We then track how many of these top-Y% clusters remain in the top-Y% on the following day (Day 2 to Day 18) and report this fraction as the

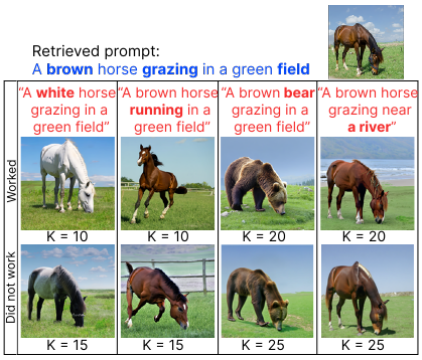


Figure 4: A noise from a brown horse successfully transforms into a white horse at K=10, while K=15 fails. The same noise becomes a bear even at K=20.

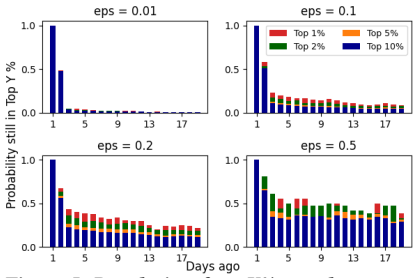


Figure 5: Popularity of top Y% popular prompt clusters over days for different similarity threshold between prompts (eps value)

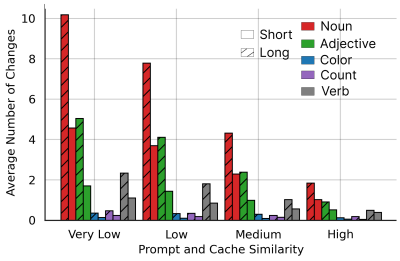


Figure 6: Average number of noun, adjective, color, count, and verb changes across similarity for both short and long prompts

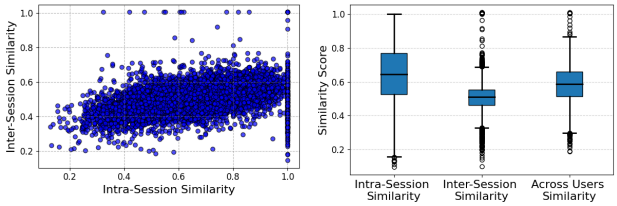
longevity probability of popular prompt clusters in Figure 5. We repeat this analysis with different *eps* values (*eps* = 0.01, 0.1, 0.2, 0.5) in DBSCAN algorithm.

Takeaway. From Figure 5, we see that most of the clusters in the top-Y% prompt clusters do not remain popular in top-Y% even on Day 2 when the clusters are tightly packed (*eps*=0.01). There is a more significant drop in popularity after 2 days. However, as we increase *eps* and make clusters more loosely packed, it effectively gathers more prompts within a cluster and hence the decay rate of popular prompts is less. We empirically verified that with increasing *eps*, higher number of prompts and more dissimilar prompts are clustered together, and hence the longevity of the top-Y% cluster increases. This shows the extent of similarity in prompts over time and the potential for reuse. *Approximate-caching* idea works even with less similar prompts, as it can recondition the retrieved noise. Thus, the high longevity of popular cluster, as seen in plots for *eps*=0.2 and *eps*=0.5, shows potential of *approximate-caching* for generating images in production environments.

(2) Most similar short and long prompt pairs

We divide the prompt queries into short and long prompts, based on the 70^{th} percentile word count (≈15 words). For each set, we then form pairs of most similar prompts using cosine-similarity between their CLIP embeddings and group them into 4 buckets based on their similarity scores: *Very Low* (less than 0.65), *Low* (0.65 to 0.8), *Medium* (0.8 to 0.9), and *High* (more than 0.9). For each bucket, we analyze what attributes (e.g., noun, adjective, verb, color, count) changed between the pair of prompts within the same bucket and show the average number of changes along these attributes, for both long and short classes of prompts in Figure 6.

Takeaway. In Figure 6, we see that as the similarity of the prompts within a pair increases, the average number of changes decreases. We also see that within each bucket, the average number of noun changes is the most, followed by adjective and verb changes. Longer prompts have more changes as compared to the shorter prompts within the same similarity bucket. This indicates that even when several attributes in the text of the prompt change, the CLIP embedding failed to appropriately distinguish the difference between the two prompts. This highlights a limitation in identifying similar prompts when prompts are very long, as two prompts can be misleadingly retrieved as very similar.



(a) *(b)*

Figure 7: Query prompt similarity (a) for inter v/s intra-session per user, (b) range for intra, inter per user and across diff users

(3) Intra vs. inter-session similarity in prompt queries

We group the prompts per user, then divide the prompts from that user into 1-hour sessions. We then compare the similarity of prompts within the same sessions (*intra*) and across different sessions (*inter*) of the *same user* in Figure 7a. In Figure 7b, we further compare these with similarity scores between prompts from sessions of 100 random *other users*.

Takeaway. Figure 7a shows, many users exhibit high intra-session similarity, meaning users tend to use similar prompts within a session with a lot of repetitions (indicated by a cluster of points at 1). Also, there is high inter-session similarity, but it is lower than intra-session, indicating diverse queries across sessions. In Figure 7b, we observe that prompts within a session (*intra-session*, left) exhibit high similarity. Notably, prompts across sessions (*inter-session*, middle) of the same user and even with prompts from different users (*across-users*, right) also demonstrate significant similarity. Interestingly, the similarity between prompts from different users is notably higher, suggesting a promising opportunity for efficient approximate caching utilizing the power of the masses.

4 NIRVANA Overview

4.1 Approximate Caching

For a new query prompt $\mathcal{P}_Q$, NIRVANA uses *approximate-caching* to reduce computation by retrieving an intermediate state that was created after K^{th} iteration of a previous image

generation process and directly reusing and reconditioning that for $\mathcal{P}_Q$ for the remaining $N-K$ steps.

Analytical Modeling: Let $\mathcal{L}$ denote the total end-to-end latency of image generation using approximate caching. Within this, $\mathcal{C}$ represents the cumulative GPU computation time for N diffusion model steps. The set of possible values for K is denoted as $\mathcal{K}$. Each search operation in the vector database (VDB) incurs a latency cost denoted as l_s, and retrieving the intermediate state from the cache introduces a latency denoted as l_r. We use f_c to indicate the overall compute savings.

Therefore, for prompts effectively utilizing *approximate caching* with a cache generated at K, the total latency experienced can be expressed as:

$$l_s + C \cdot \frac{N-K}{N} + l_r \tag{5}$$

In contrast, prompts for which NIRVANA cannot locate a match in the cache will undergo a total latency of $l_s + C$.

This distinction arises from the design, where it attempts to retrieve an intermediate state from the file system (incurring latency overhead l_r) only when a hit is confirmed in the VDB, ensuring the existence of the state in the cache.

Let us denote the *hit-rate@K* for approximate caching as $h(K)$ which is defined as the likelihood that when an intermediate state from K^{th} diffusion step is used, it takes at most $N-K$ diffusion steps to generate a faithful reconditioned image where N is fixed. That is, $(1-h(K))$ fraction of cache exists, which we cannot recondition by running $N-K$ steps.

Now at $K=0$ (running diffusion model from scratch), *all* historical prompts are theoretically usable since an image can be reconditioned in at most $N-0$ steps, leading to $h(0) = 1.0$. As K increases, $h(K)$ decreases, since we can use only a smaller fraction of intermediate states from K^{th} step to recondition an image by running diffusion at most $N-K$ steps. For lower values of K, $h(K)$ is less than 1.0 but can still be relatively high. That is, the diffusion models can effectively recondition the retrieved state if the state is from the initial diffusion steps, resulting in the generation of faithful images.

The decrease in $h(K)$ is influenced by how dissimilar the prompts are. When K surpasses a certain threshold, denoted as K_T, the retrieved state is no longer suitable for further reconditioning, as discussed in § 2.2, and thus, $h(K \geq K_T) = 0$.

Consequently, the effective fraction of savings in GPU computation for a given K can be expressed as:

$$f_C = h(K) \cdot \frac{K}{N} \tag{6}$$

It is evident that substantial savings can be achieved when both K and $h(K)$ are sufficiently high. However, the challenge lies in the fact that as K increases, $h(K)$ tends to decrease while aiming to maintain the quality $\mathbb{Q}$ of the generated images[1]. This trend is described in § 5.2 for DIFFUSIONDB dataset [79] across different discrete K values.

[1] For example, if we assume that $h(K)$ decreases linearly from 1.0 at $K=0$ to 0 at $K=K_T$ following the equation $h(K) = -\frac{K}{K_T} + 1$, then the optimal single value of K_{OPT} that maximizes fractional savings will be: $K_{OPT} = K_T/2$, resulting in effective compute savings of $f_C^{max} = K_T/4N$. Similarly, for a slowly decaying quadratic form expressed as $h(K) = -\left(\frac{K}{K_T}\right)^2 + 1$, $f_C^{max} = \frac{2K_T}{3\sqrt{3}N} > \frac{K_T}{4N}$. Therefore, the slower the decay of $h(K)$ with respect to K the higher the compute savings we can expect.

Now we define $h_{opt}(K)$ as the fraction of cache stored at K^{th} diffusion step that is used to exactly recondition an image for $N-K$ steps. For example, with $N=50, K=5$, we get $h_{opt}(K)$ is the fraction of cache that can be used to recondition an image by running diffusion steps for exact 45 steps. Thus,

$$h_{opt}(K) = h(K) - h(K'), \text{ where } \arg\min_{K'}(K' > K) \tag{7}$$

$$h(\min \mathcal{K}) = \sum_{K \in \mathcal{K}} h_{opt}(K) \tag{8}$$

In essence, $h_{opt}(K)$ quantifies the probability that K represents the maximum potential savings for incoming prompts. $h(\min \mathcal{K})$ represents the overall hit-rate, i.e., fraction of $\mathcal{P}_Q$ having a cache hit.

Our primary objective is to minimize end-to-end latency ($\mathcal{L}$) while maintaining the quality ($\mathbb{Q}$) of generated images. We operate under the constraint that reconditioning of an image with a cache at the selected K values must ensure a specified level of quality $\mathbb{Q}$ compared to when the image is generated from scratch. The goal is to find the optimal K value that satisfies these objectives and the below quality constraint.

Thus, for a given incoming prompt $\mathcal{P}_Q$ and its corresponding cached prompt $\mathcal{P}_c$

Objective (Minimize $\mathcal{L}$): (following Eq. 5)

$$\min_K \mathcal{L} = \sum_{K \in \mathcal{K}} \left(l_s + h_{\text{opt}}(K) \cdot l_r + h_{\text{opt}}(K) \cdot C \cdot \frac{N-K}{N} \right) \tag{9}$$

Quality Constraint:

$$\mathbb{Q}(I_K^c | \mathcal{P}_{c_K}, \mathcal{P}_Q) > \alpha \cdot \mathbb{Q}(I_0 | \mathcal{P}_Q) \tag{10}$$

where I_K^c represents the image generated by using cache c at K and then reconditioning for $N-K$ diffusion steps. $\alpha \in [0,1]$ represents the tolerance threshold over the quality of images generated and is such that I_K^c is not much worse than I_0. In our implementation (§ 5.3.1), we employ the CLIPScore metric [46] to define $\mathbb{Q}$ with $\alpha = 0.9$. We opt for CLIPScore due to its widespread use in evaluating image quality.

In our use case, where $l_r, l_s << \mathcal{C}$, the objective reduces to

$$\min_K \mathcal{L} = \sum_{K \in \mathcal{K}} \left(h_{\text{opt}}(K) \cdot C \cdot \frac{N-K}{N} \right) \tag{11}$$

which maximizes K and $h_{opt}(K)$ to obtain minimal latency.

This framework demonstrates the relationship between latency and quality in the context of approximate caching.

Notably, the wasted overhead of NIRVANA can be captured as $(1-h(K)) \cdot l_s$ where the vector database is queried and results in a cache miss. This means if we operate in a setting where $h(K)$ is low even if K is high, NIRVANA may not provide latency benefits if l_s is comparable to GPU compute

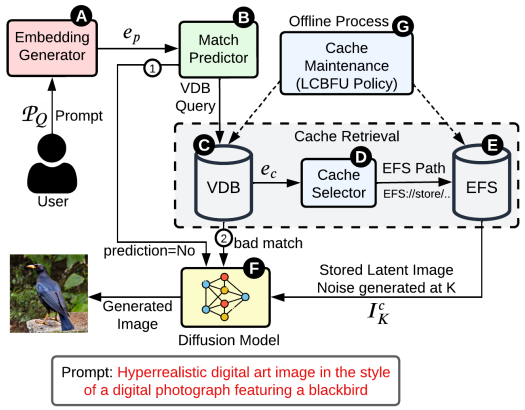


Figure 8:* NIRVANA *Overview

latency $\mathcal{C}$ and such overhead reduction can be an important design aspect as we discuss in § 5.2. It is important to note that, due to the disparate costs of high-end GPUs, even if there is not much latency reduction, it can still be significantly cost-effective to use NIRVANA as it drastically cuts expensive GPU compute costs on the cloud. In our setting, we observed l_s is of the order of 100 *ms* whereas $\mathcal{C}$ is of the order of 10 *s*.

4.2 System Components

Figure 8 shows the main components and the execution paths of NIRVANA. At first, an embedding vector e_p is generated for the current input prompt $\mathcal{P}_Q$ using an `embedding-generator` (A). Then an optional `match-predictor` module (B) predicts whether there would be a *close enough* match for this embedding in the *vector-database* (`VDB`) (C). If the presence of a similar enough cached entry is likely, then NIRVANA starts the process of retrieving an intermediate state from the cache. First, a search query is sent to the `VDB` to find the closest cached embedding of e_p, denoted by e_c. A `VDB` uses efficient approximate nearest neighbor (ANN) [62] search to find such closest embeddings. For each cached historical prompt in the `VDB`, NIRVANA stores several intermediate states during the vanilla diffusion process. Which of these intermediate states corresponding to the match prompt e_c is optimal for the new prompt vector e_p is calculated using a heuristic by the `cache-selector` module (D). Then this particular intermediate state is retrieved from `EFS` storage (E) (i.e. Elastic File System for NIRVANA) using the pointer of the storage location pointed by the search result of the `VDB` query and the particular intermediate state number (i.e. K as explained in § 4.1) calculated by the `cache-selector` module. An intermediate state I_K^c is an $L \times D$ dimensional latent representation captured during the denoising process of e_c, after K_{th} step. Finally, this retrieved intermediate state I_K^c is passed to the `DM` (F) along with e_p for it to be reconditioned for image generation for the rest of the denoising steps. However, there are two situations when NIRVANA directly falls back to the vanilla diffusion model to generate an image from scratch (using Gaussian noise), sacrificing any optimization: (i) when `match-predictor` module predicts that a close entry in the `VDB` is unlikely (arrow (1)), and (ii) when `VDB` query returns a match e_c that is very dissimilar to e_p (arrow (2)) .

To maintain the cache under a fixed size of storage and to prevent the `VDB` from arbitrarily growing and keep on storing stale entries, a `cache-maintainer` module (G) works in the offline mode and implements the novel `LCBFU` protocol to keep both the cache and `VDB` entries fresh.

5 NIRVANA Design Details

We present the detailed design of various components in NIRVANA. The full algorithm can be found in **Appendix** A.

5.1 Embedding Generator

Similar to vanilla `DMs`, NIRVANA first computes a 768-dimensional vector CLIP embedding [68] (e_p) from the text prompt. CLIP effectively positions visually similar prompts closer in the embedding space, which optimizes the likelihood of cache hits in our case.

5.2 Match Predictor

Using e_p, NIRVANA could directly make a search query to `VDB` for the closest prompt in the cache. If cache exists, there will be substantial savings in GPU usage for image generation. However, in case of *cache miss*, the search to `VDB` becomes a latency overhead without getting any reduction in the GPU computation. Now, several factors including: a) if it is in the same LAN vs. far away from the GPUs, b) the particular architecture of the `VDB` and its internal indexing mechanism, and c) the compute resources dedicated to it etc., dictate the magnitude of the wasted latency overhead during each *miss*.

To reduce this overhead, NIRVANA uses a component called `match-predictor`, ((B) in Figure 8), which *predicts* if an embedding close enough to e_p is *likely* to be present in the `VDB`. If the prediction says it is unlikely, then NIRVANA simply bypasses cache retrieval flow altogether, reducing the wasted overhead. Additionally, `match-predictor` also reduces `VDB` load corresponding to search misses, improving scalability.

Internally, `match-predictor` uses a lightweight classifier for predictions that runs on the CPU of the same node where the `DM` runs on the GPU. This reduces the classification latency by orders of magnitude compared to a query to the `VDB`, making latency overhead insignificant.

Recall from § 4.1 that analytically the latency overhead is $(1-h(K)) \times l_s$. Now let c_p denote the *precision* of the `match-predictor` classifier. The effective overhead of NIRVANA with an active `match-predictor` is then:

$$1 - max(h(K), c_p) \cdot l_s, \text{ where } h(K), c_p \in [0,1]$$

This means that either when $h(K) = 1.0$, i.e., prompts are so similar that for every incoming prompt, there is a suitable match available in the cache, or when $c_p = 1.0$, i.e., the classifier is perfect in predictions, the system will not have any wasted latency overhead. Note that, when $c_p < 1.0$, NIRVANA would miss some opportunity of compute savings during false-negative cases as it would directly fall back to vanilla `DM` and

Algorithm 1 *CacheSelector*-Profiling($\mathcal{K}$, I_K^c, α)

for K in $\mathcal{K}$ **do**
  $[I_K] \leftarrow \text{model}(\mathcal{P}_Q, I_K^c, K) \quad \forall \quad \mathcal{P}_Q$
  $\text{min\_sim} \leftarrow \min\{\text{sim } s \mid \forall I \in [I_K], \text{quality}(I) > \alpha\}$
  $\text{sim\_K\_map}[K] \leftarrow \text{min\_sim}$
end for
return sim_K_map

generate an image from scratch instead of attempting to retrieve an intermediate state. For this classifier, NIRVANA uses One-Class Support Vector Machine (One-Class SVM) [20] which constructs a decision function for outlier detection. This is trained by utilizing all prompt embeddings stored in the VDB and assigning them a positive label of 1. To achieve high precision and recall, we effectively overfit the model to the existing prompt embedding space. Further, we use Stochastic Gradient Descent (SGD) to enable faster retraining of the classifier when embeddings in VDB change significantly (i.e., > 5%). NIRVANA achieves a $c_p = 0.95$ for production prompts.

Why not a Bloom Filter? Similar to our motivation, Bloom filters [29] are popularly used as a low-cost mechanism to check cache entries in various web services and database systems [19]. Bloom-filters use hashing algorithms to calculate if an item is likely to be present in the cache with zero false negative rate but with some false positive rate. However, Bloom filters are not suitable for NIRVANA as we do not search for an *exact match* for cache entries, we search for the *nearest neighbors* in the VDB by using ANN algorithms.

5.3 Cache Retrieval

Cache Retrieval and `cache-maintainer` are the main components of NIRVANA. In this section, we elaborate on the design of the cache retrieval phase which has three internal components VDB, `cache-selector`, and storage-system. In § 5.4 we discuss the design of `cache-maintainer`.

Vector Database: NIRVANA stores the embedding of the historical prompts in a vector-database (VDB) for fast and efficient similarity search with the embedding of incoming prompts. A VDB uses indexing methods like quantization, graphs, or trees to store and perform high-dimensional similarity search over vectors. For a query embedding vector, it can find m approximate nearest neighbors. In NIRVANA, for each incoming search with e_p, VDB already populated with the embeddings, its payload points to the path where the corresponding intermediate states of e_c at different Ks are stored. NIRVANA uses `cosine-similarity` as a measure to find the nearest neighbor. While NIRVANA can work with any VDB such as `Qdrant` [21], `Milvus` [18], `Weaviate` [23], `Elasticsearch` [13], in this paper we present results with `Qdrant` as benchmarking [3] shows low read latency at scale and moderately low update and delete latency.

Elastic File System as Storage: NIRVANA stores the actual intermediate noise states on AWS Elastic File System (EFS). After comparing with LustreFS [16] on read latency, throughput, and storage cost, we chose EFS as our storage system.

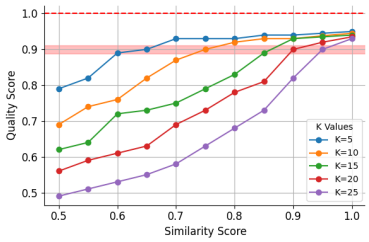


Figure 9: Quality of image generation with cache vs. similarity score across K. Thresholds for cache usage at different K values.

```
def cache_selector(s):
  if s > 0.95: k = 25
  elif s > 0.9: k = 20
  elif s > 0.85: k =
     15
  elif s > 0.75: k =
     10
  elif s > 0.65: k = 5
  else: k = 0
  return k
```

Figure 10: Determining K based on similarity score (s).

The size of the files containing intermediate-state depends on the architecture of the DM. For NIRVANA the default DM uses an intermediate-state of the size of 144 KB and stores for 5 distinct values of $K \in \mathcal{K} = \{5, 10, 15, 20, 25\}$ as increasing beyond $K = 25$ compromises output quality significantly and more granular Ks does not necessarily yield proportional gains but increases algorithmic complexity.

5.3.1 Cache Selector

The `cache-selector` component primarily determines which $K \in \mathcal{K}$ for e_c from EFS should be retrieved to recondition the intermediate state for the rest of the $N - K$ steps with prompt $\mathcal{P}_Q$ for maximum compute saving while maintaining acceptable image quality.

How do we choose the K? We observed empirically that the number of steps that can be skipped is correlated to the similarity between e_p and e_c. Based on this observation, we perform an offline profiling/characterization (Algorithm 1) to find the appropriate K such that the intermediate-state generated at K^{th} diffusion step is optimal based on the similarity score between the two prompt's embeddings (Eq. 10). The algorithm ("*CacheSelector*-Profiling(.)") generates images at each value of K for a set of prompts with their nearest cache prompt. It then finds the minimum similarity score such that all generated images are above a quality threshold α. This is then chosen as the minimum similarity at which the value of K works and is stored to be used later at run-time. As shown in Figure 9, we plot the image quality (higher the better) against various similarity scores between the query and the cached prompts across multiple values of K.

We use the profiled information to determine the optimal value of K for image generation at run-time. Using the similarity score s between e_p and e_c obtained from VDB, we choose a K such that the quality score at the plot $(x = K, \textit{similarity\_score} = s)$ is $\geq \alpha$. We choose $\alpha = 0.9$. In simple words, we search the inverse map stored by Algorithm 1. The red band in Figure 9 represents the threshold α. Each line in the figure, corresponding to different K values, intersects this red band at the specific similarity scores above which that particular K value is applicable. Thus, if similarity score is high, we can skip a greater number of diffusion steps and choose higher K to achieve acceptable image quality. But, if similarity score is low, we can skip only a limited number of iterations and hence, a lower K. Figure 10 shows the logic

for computing the optimal K given a similarity score s between two prompts that is identified for DiffusionDB dataset using the previously discussed profiling technique. Based on this, we query the cache storage to obtain I_K^c. Once NIRVANA retrieves the appropriate intermediate noise from the cache , DM denoises it for N-K steps to generate the final image.

Quality vs. performance tradeoff: NIRVANA is designed to extract as much compute savings as possible without degrading the quality of images. However, if more aggressive compute savings is needed for certain use cases that are ready to sacrifice some accuracy, NIRVANA can trivially expose a *knob* that can be used to trade off quality vs. compute savings. The cache-selector heuristic can be biased to select higher values of K. This will provide more compute reduction while letting image quality degrade, as NIRVANA will get fewer steps to recondition the retrieved noise according to the new prompt. At the extreme, for $K = N$, NIRVANA will behave exactly like a pure image retrieval system [4, 7].

5.4 Cache Maintenance: LCBFU Policy

In popular production text-to-image system, there is a large stream of incoming prompts. The cost associated with total storage used increases, and VDB performance also degrades beyond a certain limit of entries if we keep on storing the cache. Therefore, even though NIRVANA can theoretically support infinite cache, the storage cost and increasing search latency would eventually make it unattractive. The cache-maintainer component works in the background to maintain the entries in the cache storage in EFS and in VDB.

To achieve this, NIRVANA uses a novel cache maintenance policy *Least Computationally Beneficial and Frequently Used* (LCBFU) customized for approximate caching in DMs. The fundamental idea is that not all intermediate states are equally beneficial for overall compute and latency savings. Intermediate states with high K values can provide huge compute savings but can be used only when $\mathcal{P}_Q$ has high similarity to one of the cached items. Items stored at low K are usable with a variety of prompts since they can be reconditioned even when the similarity with $\mathcal{P}_Q$ is relatively low, but provides low compute savings. With LCBFU, NIRVANA can limit a total of 1 TB cache in EFS without degrading any performance or quality, which on average corresponds to 1.5 million intermediate-states or noises and 300k unique prompt embeddings stored in the VDB.

Drawbacks of traditional cache policies: Due to the unique compute model of approximate-caching, the traditional cache policies such as LRU (least recently used), LFU (least frequently used), and FIFO (first-in first-out) are not very useful in NIRVANA, since they treat each item in the case homogeneously and only focus on the access patterns or arrival sequence while evicting an item. As discussed in § 2.1, hit-rate alone does not determine the efficiency of NIRVANA as items with large K values, even though less frequently accessed can provide significant overall compute reduction.

LCBFU: We design the LCBFU that takes into account both access frequency of items as well as potential compute benefit in case of a *hit*. It evicts items with least *LCBFU-score* which for each item i is calculated as $f_i \times K_i$. Here f_i is the access frequency of item i and K_i denotes which step of the denoising process this intermediate state belongs to. Notably, since the cache-selector heuristic determines K based on the similarity of e_p and e_c, an aggressive heuristic (high K for low similarity), will force high f_i for high K noises and hence low K noises will be evicted. For example, an item with $K_i = 25$ was accessed 100 times, its LCBFU-score is 2500, while an item with $K_j = 5$ was accessed 200 times, its LCBFU-score is 1000. Thus, the item i has a higher LCBFU-score since it will provide better compute savings while generating an image. LCBFU prioritizes compute savings while managing the trade-offs for optimal user experience at different K values.

The complete mechanism of LCBFU is described as follows:

- **Insertion:** With every cache miss, we directly insert all the intermediate states generated at diffusion denoising step $K \in \mathcal{K}$ to the cache-storage and the corresponding embedding of the prompt to the VDB. Thus $|\mathcal{K}|$ are stored in the cache storage per prompt. The insertions are performed without any eviction until we reach the target storage limit. After that, every insertion is preceded by an eviction.
- **Eviction:** For eviction, LCBFU maintains a running list of *LCBFU-score* in a K-min heap, and evict the top-$|\mathcal{K}|$ items from the heap root just before inserting $|\mathcal{K}|$ intermediate-states for a new prompt. The LCBFU-score evicts image noises which contribute least to compute savings.

With this cache eviction policy, cases can arise for a particular prompt where noises at some Ks are evicted, while noises at other Ks are still in the cache. This creates ***holes*** in the intermediate-states stored.

Handling of holes: Once eviction policy creates a hole, no straightforward way exists to fill that hole, as targeted regeneration of the intermediate state is not possible without running the full diffusion process. However, the heuristic used by cache-selector (§ 5.3.1) is oblivious to the existence of these holes while determining an appropriate K to be used with the retrieved prompt. NIRVANA handles this situation by choosing the intermediate state with the largest value K that is less than or equal to the optimal K (e.g. it can use K=10, if a hole is at K=15). This ensures that NIRVANA continues to generate high-quality images, albeit with little sacrifice in potential compute savings when it encounters the holes. However, we observed that such cases arise only in **4-5%** of the prompts, resulting in imperceptible performance degradation.

NIRVANA does not actively use any strategy to clean such holes. Only when all the intermediate-states for a prompt corresponding to all the $|\mathcal{K}|$ values turn into holes, LCBFU marks that prompt embedding as dirty and removes it from VDB as well as corresponding metadata from the storage-system.

cache-maintainer performs both insertions and deletions on VDB in batches, and at the same time, the classifier in the

match-predictor is also retrained with the fresh entries in the VDB. Recall, as mentioned in § 5.2 and § 5.3, both VDB update and classifier updates are fast and takes only around 7.5 and 0.04 seconds respectively for 10k records.

Discussions: In **Appendix** C, we explore NIRVANA's adaptability to image diversity, workload shifts, the generalizability of caching, LCBFU policy, and its use with ML optimizations.

6 Implementation

NIRVANA is implemented in Python using PyTorch [65] for diffusion model architecture, enabling user-friendly modules and seamless support for SYSTEM-X production integration. Our design also enables easy integration with image generation frameworks like Stable Diffusion. More details on batching and framework used are given in **Appendix** B

System Components: Our system components consist of (i) a classifier that uses *SGDOneClassSVM* [20] from scikit-learn [66] with $\nu = 0.001$, where ν controls the trade-off between training errors and support vectors, (ii) VDB hosted in a Docker container on an AWS m5.4xlarge EC2 instance with HNSW indexing of 256 for prompt embedding search [62], (iii) AWS EFS system that offers web and file system access, object storage, and scalability, (iv) MySQL database to record accesses of cache items, and (v) a text-to-image generator powered by a larger version of stable diffusion-based model with DDIM (Denoising Diffusion Implicit Model) sampler [76]. It generates images in $N = 50$ iterations utilizing approximately 8 GB of memory for a batch size of 1.

7 Evaluation

We first evaluate the overall effectiveness of NIRVANA in terms of quality and in providing significant compute and end-to-end latency savings, throughput, and cost savings for serving on the AWS cloud platform. We perform a user study with 60 participants to compare the image generation quality against baselines. We present LCBFU benefits against common cache-management policy to demonstrate the effectiveness of NIRVANA's internal component design. We also perform and present ablation studies. The key takeaways are:

- NIRVANA generates **high quality** image while **reducing** both GPU usage and end-to-end latency by up to **50%**.
- NIRVANA **reduces** all three - **cost**, **latency** and **compute** requirements of DM by **∼20%** on average.
- With a **27% improvement** in system **throughput**, NIRVANA ensures a stable user experience with minimal response time variations.

7.1 Methodology

Base Diffusion Model: As the base text-to-image model, we use a larger Stable Diffusion based model (which we call **VANILLA**), having approximately 1.5 times the number of parameters as compared to the 2.3B parameter Stable Diffusion XL model [67]. Our model operates within a 96-pixel latent space and performs $N = 50$ denoising steps to generate an image in 8.59 seconds, on average, on an A10g GPU. The final image generated is of size $768 \times 768 \times 3$.

Experimental Setup: We run the DM and embedding-generator of NIRVANA on a single NVIDIA A10g GPU (24 GB GPU memory), while the other components are run on a 32-core AMD EPYC 7R32 2.8 GHz CPU with 128 GB CPU memory. The GPU and the CPU machines were attached to a sub-network which included EFS and VDB. We optimize GPU usage by reusing prompt embeddings with diffusion model embeddings, reducing GPU overhead.

Dataset: We evaluate NIRVANA on two production datasets.

- **DiffusionDB** [79]: A total of 2M images for 1.5M unique prompts, with a total dataset size of $\approx 1.6\ TB$. We filtered the dataset and removed NSFW images
- SYSTEM-X: Prompts from production setup SYSTEM-X spanning over 8 weeks, containing over 7M images for 6.2M unique prompts, with a total size of $\approx 5\ TB$.

Unless otherwise mentioned, we evaluate the results on DIFFUSIONDB dataset, since SYSTEM-X data is proprietary.

Baselines: We compare two versions of NIRVANA: (1) with match-predictor (referred as NIRVANA) and (2) *without* match-predictor (referred as NIRVANA-w/oMP) against the following baselines:

- **GPT-CACHE**: Retrieves image for the closest prompt based on BERT embedding similarity. Otherwise, generates an image from scratch [7]
- **PINECONE**: Retrieves image for the closest prompt based on CLIP text embedding similarity. Otherwise, generates an image from scratch [4]
- **CRS (Clip Retrieval System)**: Clip Retrieval System [5] is another image retrieval method that uses the embedding of the final image generated by the previous prompts when retrieving the closest image for a given input prompt.
- **SMALLMODEL**: A smaller diffusion model [69] with 860M parameters, consuming only 33% of compute/latency compared to VANILLA, generating an image by 50 diffusion steps in 3.05 seconds on average on A10g.

Workload Generator and Cache Preload: The workload generator dispatched prompts in the order in which they arrived, using the dataset's arrival timestamp field to create a stream of prompts. Each prompt query was dispatched to the NIRVANA immediately upon the completion of the preceding query. This query stream began only after the initial 10k prompts were employed to preload the VDB and EFS with their respective caches for kick-starting the system.

Evaluation Metrics: We evaluate NIRVANA on various metrics covering both quality and efficiency aspects.

1. **Quality Metrics:**
   - **FID Score (Fréchet Inception Distance):** Computes the difference between two image datasets and correlates with human visual quality perception [47].
   - **CLIP Score:** Evaluates the alignment between generated images and their textual prompts [46]

- **PickScore:** A metric designed for predicting user preferences for generated images [53]

2. **Efficiency Metrics:** To evaluate the efficacy of NIRVANA in providing system efficiency, we report an average of 5 runs for % savings in GPU usage time, % reduction in end-to-end latency of image generation, % increase in throughput as number of images generated per second in a cluster and also amortized dollar-cost per image generation. We also report *hit-rate* $= h(min\mathcal{K})$ (Eq. 8) for NIRVANA.

7.2 Overall Performance on Quality

In this section, we evaluate how NIRVANA performs significantly better with respect to image generation quality.

Quantitative Generation Performance: Table 1 summarizes NIRVANA's improvements in terms of the quantitative image quality metrics. When compared against the retrieval-based baselines (§7.1), NIRVANA and NIRVANA-w/oMP improve performance significantly as captured by all three metrics [77]. Retrieval-based baselines directly retrieve the image generated from the most similar previous prompt for the query prompt. Hence, these methods fail to capture the differences between the incoming prompt and the retrieved prompt, since similarity metrics are unable to capture these (§3). Thus, all the retrieval-based baselines incur a significant hit in the quality of the image generated, which is of utmost importance in production user-facing text-to-image use cases. Compared to SMALLMODEL, both NIRVANA and NIRVANA-w/oMP perform far superior, which shows bigger models retain quality even with *approximate-caching*.

The metrics for VANILLA show the generated image quality without any kind of approximations. The performance of NIRVANA is very close to VANILLA when compared against CLIP and Pick score for both datasets. This shows that *approximate-caching* does not hurt the overall performance.

We measure FID (lower the better) of baselines against images generated using VANILLA for the same prompts. We also compute FID of VANILLA to indicate the inherent variability of the generated images without any change in the base model. To calculate FID for VANILLA, we generated 4 sets of images with 4 different seeds and calculated the FID between these resulting ${}^4C_2$ sets of images. As can be seen for both datasets, NIRVANA and NIRVANA-w/oMP exhibit much lower FID values than even the internal dissimilarities between different sets of generations by the VANILLA model. This means that images generated by NIRVANA will be indistinguishable from the VANILLA model - which is the design goal.

NIRVANA performs slightly better than NIRVANA-w/oMP due to the presence of `match-predictor` which generates an image from scratch during a predicted cache miss. However, this comes with a very small hit on NIRVANA's efficiency, compared to NIRVANA-w/oMP (see §7.3).

User Study for Accessing Quality: We conduct a user study with 60 participants to demonstrate the qualitative analysis. We evaluate 1000 randomly chosen prompts from DIFFUSIONDB, where each user was presented with 15 random `<prompt, generated-image>` pairs and were asked to vote `Yes`, or `No` based on whether the generated image properly represents the given prompt. Out of these 15 prompt-image pairs, 5 pairs were generated by GPT-CACHE, the best-performing image-retrieval-based baseline, next 5 pairs generated by NIRVANA and, next 5 using VANILLA. The images from these three types were shuffled and presented to each user in a random order and no prompt was repeated within a session. Users were also given the option to disclose reasons for `No`.

Dataset	Models	Quality		
		FID ↓	*CLIP Score* ↑	*PickScore* ↑
DiffusionDB	GPT-CACHE	7.98	25.84	19.04
	PINECONE	10.92	24.83	18.92
	CRS	8.43	24.05	18.84
	SMALLMODEL	11.14	25.64	18.65
	NIRVANA − *w/o*MP	4.94	28.65	20.35
	NIRVANA	**4.68**	**28.81**	**20.41**
	VANILLA	6.12-6.92	30.28	20.86
SYSTEM-X	GPT-CACHE	8.15	26.32	19.11
	PINECONE	10.12	24.43	18.83
	CRS	8.38	23.81	18.78
	SMALLMODEL	11.35	25.91	18.92
	NIRVANA − *w/o*MP	4.48	28.94	20.31
	NIRVANA	**4.15**	**29.12**	**20.38**
	VANILLA	5.42-6.12	30.4	20.71

Table 1: We compare* NIRVANA *against several baselines* GPT-CACHE, PINECONE, CRS *that are pure retrieval-based techniques, a smaller-model, and with vanilla diffusion model. CLIP score and PickScore are based on text-to-image score. FID is image-to-image comparisons, using* VANILLA *images as Ground Truth. Classifier has 0.96 Precision, 0.95 Recall.

In Figure 11, we show the ratio between `Yes` and `No` responses for each of the three models aggregated across all users. It can be seen that GPT-CACHE gets the lowest number of `Yes` votes, while NIRVANA is just marginally beneath the upper bound VANILLA. When analyzed for reasons of `No` (∼12% of the negative responses), we discovered that 80% of these reasons were for the images generated by GPT-CACHE. These insights underscore the challenges of relying solely on retrieval-based solutions for image generation and hence we do not use these baselines in further evaluation.

This reinforces the fact that NIRVANA *is much superior in maintaining image quality, which is of paramount importance for commercial deployment.*

7.3 System Efficiency of NIRVANA

We compare NIRVANA in terms of image generation latency, hit rate, compute savings, and cost of running NIRVANA.

Latency: Latency in Table 2 refers to the average end-to-end latency across two million prompt queries tested. We plot the n^{th} percentile over the median of response latencies for some baselines. An image generation system should exhibit low latency for enhanced user experience. However, pure retrieval techniques like GPT-CACHE provide low latency but produce low-quality images and experience significant fluctuations in their end-to-end latency since they need to

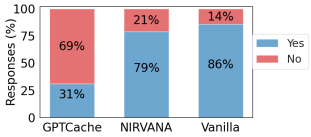


Figure 11: User survey response

	Latency(s)	90^{th}/median	95^{th}/median	99^{th}/median
GPTCache	2.8	29.52	29.64	29.75
NIRVANA	6.9	1.20	1.21	1.21
VANILLA	8.6	1.01	1.02	1.02

Table 2: Average latency and n^{th} percentile over median values of the response latencies for approaches

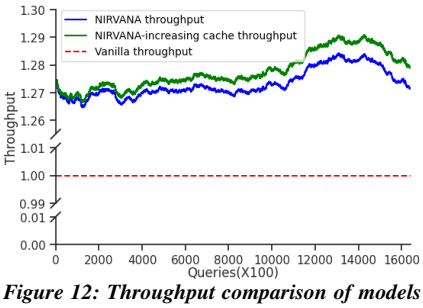


Figure 12: Throughput comparison of models against a stream of queries over time

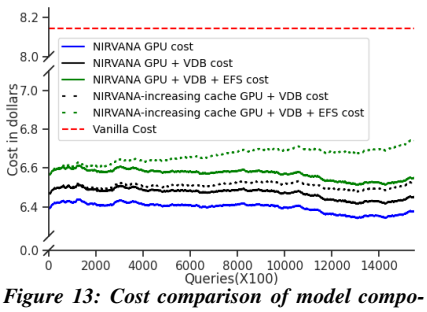


Figure 13: Cost comparison of model components against a stream of queries over time

run diffusion model from scratch for certain prompts that are dissimilar from the cached prompts, thus impacting the user experience [26] as shown in Table 2. NIRVANA reduces such variance in latency compared to the baselines and also the overall latency compared to VANILLA providing a much more stable and faster user experience. The key to minimizing response time variability lies in the smoother transition in compute time across different prompts, attributable to NIRVANA's ability to retrieve caches at various values of K.

This observation underscores that NIRVANA *not only reduces overall latency by 19.8% but also minimizes variance in response times across different prompts, ultimately providing a consistent and stable user experience [30].*

Throughput: We quantify throughput as the number of prompts processed per minute by the system. To evaluate this, we replay the stream of prompts from the DiffusionDB dataset. We assess the relative throughput of NIRVANA in comparison to VANILLA, considering two cache settings. The first setting employs LCBFU with a cache size of 1.5 million items, while the second setting involves an increasing cache configuration where no cache eviction occurs, effectively providing a theoretically infinite cache size. Figure 12 represents our findings, where the x-axis corresponds to the stream of queries, and we plot the relative throughput of the system. Notably, our results indicate that NIRVANA achieves ∼1.28 times higher throughput than VANILLA for both settings.

Cost of Image Generation: To provide a detailed cost breakdown for NIRVANA and VANILLA, we considered specific AWS components and their associated expenses, as per AWS pricing [1]. We use the g5.24xlarge GPU (96 GB GPU) instance from the US East region which costs $8.144 per hour. This cost was used to estimate the GPU-related expenses incurred by VANILLA and NIRVANA. VANILLA solely uses the GPU resources for each image generation. However NIRVANA also relies on additional components besides GPU, namely VDB [6] and EFS. For VDB, the cost is estimated at $0.12 per hour which covers storage and search operations performed over the prompt embeddings. Additionally, NIRVANA incurs costs associated with EFS. In the US East region, utilizing the standard storage type with 20% frequent access over elastic throughput cost amounts to $0.09 per hour. To determine the overall amortized cost of NIRVANA, we calculated it by dividing the GPU cost by the throughput and then adding the costs related to VDB and EFS. It is important to note that this cost analysis was conducted under identical settings for both systems. Despite the inclusion of these additional expenses, Figure 13 highlights that NIRVANA manages to achieve a remarkable 19% reduction in cost compared to VANILLA. This significant cost efficiency underscores the practical advantages of NIRVANA in real-world deployment.

Hit-Rate and Compute Savings: Figure 15 highlights the *hit-rate* and *compute-savings* of NIRVANA across K (x-axis). Averaging over all Ks, NIRVANA achieves a substantial *hit-rate* of 88% and noteworthy *compute savings* of 21% as indicated by the blue and black dotted lines. The cumulative *hit-rate* curve illustrates how the *hit-rate* varies with different values of K and plots $h(K)$. Additionally, the bar plot showcases the potential savings achievable at each K and the actual savings realized at specific K values. For instance, at $K = 25$, there is a potential savings of 50%, while the actual *hit-rate* is 8%, resulting in an actual savings of 4%.

7.4 LCBFU Performance

We experiment the effectiveness of LCBFU against common caching techniques like FIFO, LRU, and LFU. To ensure a fair evaluation, we maintained the same workload generator settings across all experiments with cold cache setting. FIFO removed the earliest added noises from the cache. LRU and LFU evict cache items based on their access frequency and recency. Table 3(a) reports *hit-rate* (Eq. 8) and % compute savings (% of GPU time savings when compared against VANILLA) for various cache eviction policies across different cache sizes. The *hit-rate* of LCBFU is comparable to LRU and LFU while outperforming FIFO. Notably, as highlighted in Table 3(b), the proposed LCBFU offers substantial compute savings compared to all other policies since it is designed to incorporate K with image access frequency for eviction. LCBFU is not designed to have the best hit rate.

7.5 Sensitivity Analysis

We now present some sensitivity analysis regarding NIRVANA's design choices. Additional sensitivity analyses are present in the **Appendix** D.

Embedding type: We conducted a comparison of NIRVANA using various types of embeddings for query and/or cache. Some embeddings are applied to the query as well as cache prompts, while other is applied just to the cached

Cache Size	#noises in cache	*FIFO*	*LRU*	*LFU*	LCBFU
1*GB*	1500	0.58	0.65	0.64	0.65
10*GB*	15000	0.68	0.77	0.78	0.77
100*GB*	150000	0.74	0.85	0.83	0.82
1000*GB*	1500000	0.83	0.95	0.94	0.93

(a) Hit rate

Cache Size	#noises in cache	*FIFO*	*LRU*	*LFU*	LCBFU
1*GB*	1500	0.11	0.12	0.12	0.12
10*GB*	15000	0.13	0.14	0.14	0.15
100*GB*	150000	0.14	0.16	0.16	**0.18**
1000*GB*	1500000	0.17	0.20	0.19	**0.23**

(b) % Compute savings

Table 3: Performance of different eviction techniques. Compute savings with `LCBFU` eviction is significant.

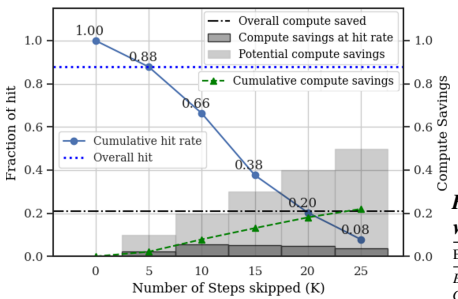


Figure 15: Hit rate and compute saved across K's

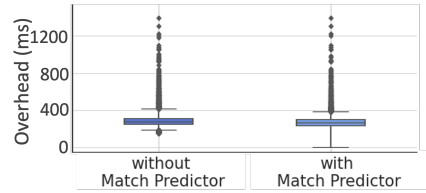


Figure 16: Overhead latency distribution with and without `match-predictor`.

Embedding	Prompt/Cache	FID↓	CLIPScore↑	HitRate↑	ComputeSavings↑
BERT	Query & Cache	4.53	28.94	0.77	0.16
CLIPImage	Only Cache	4.85	28.49	0.84	0.18
CLIP	Query & Cache	4.94	28.65	**0.93**	**0.23**

Table 4: Quality of generation for different embeddings.

prompt and CLIP embedding is maintained for the query prompt. This is indicated by the column 'Prompt/Cache'. The results, summarized in Table 4, indicate that the quality of image generation, assessed through metrics such as FID, CLIP Score, and PickScore remains consistent across different embedding types. However, the hit rate and compute savings achieved with the CLIP Text embeddings significantly outperform the other two embedding types. Hence, we selected CLIP Text embeddings for our system design.

Effectiveness of Match-Predictor: We conducted an evaluation to assess the effectiveness of using the `match-predictor` within NIRVANA, where we measured the average overhead in image generation latency. Figure 16 shows that `match-predictor` contributes significantly to lowering the overhead latency values as a considerable fraction of queries has negligible overhead, approximately equal to zero. This behavior is attributed to the `match-predictor`'s ability to promptly predict whether a particular prompt is present in the cache or not. It reduces the requirement for I/O-related activities, resulting in decreased latencies. Network delays in `VDB`/`EFS` cause right-tail latency. About 4% of these cases result from cache misses, reduced to zero with `match-predictor`. However, right-tail latency persists due to network call delays, even with `match-predictor`.

8 Related Works

ML optimizations. Various techniques like model distillation [63, 72], pruning [38], quantization [60] and others [50, 57, 85] exists for large `DMs` but often degrades the quality as well. DeepSpeed [35] is an optimization library for distributed inference and implements multiple techniques for the same. These can make the base model faster but are orthogonal to NIRVANA as they do not fundamentally change the nature of the iterative denoising process that NIRVANA exploits for compute reduction.

Model-serving in Cloud. Past research [33, 44, 58, 83, 86] explored efficient ML model serving to reduce inference latency. Clipper [33] implements optimizations like layer caching, ensembling methods, and straggler mitigation. Cocktail [44] designs a cost-effective ensemble-based model serving framework along with proactive autoscaling for resource management to provide high throughput and low latency. Other works [31, 32, 86] optimized the prediction pipeline cost during load variations. However, these are complimentary to NIRVANA and can be integrated easily. Tabi [78] uses multiple heterogeneous models for Large Language Models.

Text-to-image models. Diffusion model is one of the main classes of text-to-image models, which was popularized by Dall-E [12], Imagen [71] and Stable Diffusion [69]. Several other enterprises like Midjourney [17], Deci [15], and Adobe [9] have built their own diffusion models. Algorithmic optimizations include designing of faster sampling step [49, 54, 76, 80, 90] and parallel sampling [75, 90] and hence trading off compute for speed. However, our work is orthogonal, and the underlying diffusion model can be chosen from any of the above-mentioned works.

Caching. Caching in DNN inference has been explored in the past [33, 43, 55, 56] including caching intermediate layer outputs to avoid running every layer again on different input [33, 56]. Kumar *et al.* [56] coins the term *approximate caching* for above, but our semantics are orthogonal since we cache the intermediate image noise, and not the model layer outputs. None of the caching policies implemented in these works (LRU, static cache) work in our case. Other caching techniques [28, 84] are non-trivial to extend for our purpose. Retrieval-based works [4, 5, 7] uses caching to retrieve images for the most similar prompt, but suffer in quality (see §7).

Approximations in System Design: Various forms of approximations are employed in system designs to reduce redundant computation, enhancing efficiency in domains such as big data [24, 25, 27, 40, 41, 48, 64, 74], mobile computing [45, 81], and video processing [82, 88]. However, none of these approaches apply to `DMs`.

9 Conclusion

In this paper, we introduced the design and implementation of NIRVANA that uses a novel technique called *approximate-caching* to significantly reduce compute cost and latency during text-to-image generation using `DMs` by caching and reusing intermediate-states created while processing prior text prompts. We also presented a new cache management technique to optimize performance under a fixed storage.

References

[1] Aws product and service pricing | amazon web services. https://aws.amazon.com/pricing/?aws-products-pricing.sort-by=item.additionalFields.productNameLowercase&aws-products-pricing.sort-order=asc&awsf.Free%20Tier%20Type=*all&awsf.tech-category=*all. (Accessed on 09/20/2023).

[2] Complete guide to samplers in stable diffusion - félix sanz. https://www.felixsanz.dev/articles/complete-guide-to-samplers-in-stable-diffusion. (Accessed on 09/21/2023).

[3] Hugging face model repository. https://huggingface.co/models.

[4] Making stable diffusion faster with intelligent caching | pinecone. https://www.pinecone.io/learn/faster-stable-diffusion/. (Accessed on 09/21/2023).

[5] rom1504/clip-retrieval: Easily compute clip embeddings and build a clip retrieval system with them. https://github.com/rom1504/clip-retrieval. (Accessed on 09/21/2023).

[6] Vector search database | qdrant cloud. https://cloud.qdrant.io/calculator. (Accessed on 09/20/2023).

[7] zilliztech/gptcache: Semantic cache for llms. fully integrated with langchain and llama_index. https://github.com/zilliztech/GPTCache. (Accessed on 09/21/2023).

[8] Adobe express with ai-powered firefly integration now commercially available. https://news.adobe.com/news/news-details/2023/Adobe-Express-With-AI-Powered-Firefly-Integration-Now-Commercially-Available/default.aspx, 2023.

[9] Adobe firefly. https://www.adobe.com/sensei/generative-ai/firefly.html. 2023.

[10] Adobe unleashes new era of creativity for all with the commercial release of generative ai. https://news.adobe.com/news/news-details/2023/Adobe-Unleashes-New-Era-of-Creativity-for-All-With-the-Commercial-Release-of-Generative-AI/default.aspx, 2023.

[11] Aitemplate. https://github.com/facebookincubator/AITemplate, 2023.

[12] Dall-e 2. https://openai.com/dall-e-2. 2023.

[13] Elasticsearch. https://www.elastic.co/, 2023.

[14] Intel labs introduces ai diffusion model, generates 360-degree images from text prompts. https://www.businesswire.com/news/home/20230621842353/en/Intel-Labs-Introduces-AI-Diffusion-Model-Generates-360-Degree-Images-from-Text-Prompts, 2023.

[15] Introducing decidiffusion 1.0: : 3x the speed of stable diffusion with the same quality. https://deci.ai/blog/decidiffusion-1-0-3x-faster-than-stable-diffusion-same-quality/. 2023.

[16] Lustrefs. https://www.lustre.org/, 2023.

[17] Midjourney. https://www.midjourney.com/home/. 2023.

[18] Milvus - vector database. https://milvus.io/, 2023.

[19] Myrocks and bloom filters. https://mariadb.com/kb/en/myrocks-and-bloom-filters/, 2023.

[20] Oneclass svm. https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html, 2023.

[21] Qdrant - vector database. https://qdrant.tech/, 2023.

[22] Stable diffusion batch prediction with ray data. https://docs.ray.io/en/latest/data/examples/stablediffusion_batch_prediction.html, 2023.

[23] Weaviate - vector database. https://weaviate.io/, 2023.

[24] Shubham Agarwal, Gromit Yeuk-Yin Chan, Shaddy Garg, Tong Yu, and Subrata Mitra. Fast natural language based data exploration with samples. In *Companion of the 2023 International Conference on Management of Data*, pages 155–158, 2023.

[25] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. {GRASS}: Trimming stragglers in approximation analytics. In *11th USENIX symposium on networked systems design and implementation (NSDI 14)*, pages 289–302, 2014.

[26] Xiao Bai, Ioannis Arapakis, B Barla Cambazoglu, and Ana Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Transactions on Information Systems (TOIS)*, 36(2):1–42, 2017.

[27] Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, Thorsten Strufe, et al. {PrivApprox}:{Privacy-Preserving} stream analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 659–672, 2017.

[28] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. {LHD}: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[29] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[30] Peter M Broadwell. *Response time as a performability metric for online services*. Computer Science Division, University of California, 2004.

[31] Lequn Chen, Weixin Deng, Anirudh Canumalla, Yu Xin, Matthai Philipose, and Arvind Krishnamurthy. Symphony: Optimized model serving using centralized orchestration. *arXiv preprint arXiv:2308.07470*, 2023.

[32] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: Ml prediction pipeline provisioning and management for tight latency objectives, 2020.

[33] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[34] Florinel-Alin Croitoru, Vlad Hondru, Radu Tudor Ionescu, and Mubarak Shah. Diffusion models in vision: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[35] deepspeed.ai. Deepspeed. https://www.deepspeed.ai/, 2023.

[36] Prafulla Dhariwal and Alexander Nichol. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems*, 34:8780–8794, 2021.

[37] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[38] Gongfan Fang, Xinyin Ma, and Xinchao Wang. Structural pruning for diffusion models. *arXiv preprint arXiv:2305.10924*, 2023.

[39] Paul A Gagniuc. *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[40] Shaddy Garg, Subrata Mitra, Tong Yu, Yash Gadhia, and Arjun Kashettiwar. Reinforced approximate exploratory data analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 7660–7669, 2023.

[41] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 383–397, 2015.

[42] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2014.

[43] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. In *International Conference on Learning Representations*, 2017.

[44] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1041–1057, 2022.

[45] Peizhen Guo, Bo Hu, Rui Li, and Wenjun Hu. Foggycache: Cross-device approximate computation reuse. In *Proceedings of the 24th annual international conference on mobile computing and networking*, pages 19–34, 2018.

[46] Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. Clipscore: A reference-free evaluation metric for image captioning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7514–7528, 2021.

[47] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[48] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries! *arXiv preprint arXiv:1909.00607*, 2019.

[49] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[50] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

[51] Jina.ai. Benchmark vector search databases with one million data. https://jina.ai/news/benchmark-vector-search-databases-with-one-million-data/, 2022.

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

[53] Yuval Kirstain, Adam Polyak, Uriel Singer, Shahbuland Matiana, Joe Penna, and Omer Levy. Pick-a-pic: An open dataset of user preferences for text-to-image generation. *arXiv preprint arXiv:2305.01569*, 2023.

[54] Zhifeng Kong and Wei Ping. On fast sampling of diffusion probabilistic models. In *ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models*, 2021.

[55] Roland Kuhn. Speech recognition and the frequency of recently used words: A modified Markov model for natural language. In *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*, 1988.

[56] Adarsh Kumar, Arjun Balasubramanian, Shivaram Venkataraman, and Aditya Akella. Accelerating deep learning inference via freezing. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[57] Fan Lai, Yinwei Dai, Harsha V Madhyastha, and Mosharaf Chowdhury. {ModelKeeper}: Accelerating {DNN} training via automated training warmup. In *NSDI*, 2023.

[58] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *OSDI*, 2018.

[59] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[60] Xiuyu Li, Yijiang Liu, Long Lian, Huanrui Yang, Zhen Dong, Daniel Kang, Shanghang Zhang, and Kurt Keutzer. Q-diffusion: Quantizing diffusion models. *arXiv*, 2023.

[61] Luping Liu, Yi Ren, Zhijie Lin, and Zhou Zhao. Pseudo numerical methods for diffusion models on manifolds. *arXiv preprint arXiv:2202.09778*, 2022.

[62] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[63] Chenlin Meng, Robin Rombach, Ruiqi Gao, Diederik Kingma, Stefano Ermon, Jonathan Ho, and Tim Salimans. On distillation of guided diffusion models. In *CVPR*, 2023.

[64] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476, 2018.

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[66] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[67] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. Sdxl: improving latent diffusion models for high-resolution image synthesis. *arXiv preprint arXiv:2307.01952*, 2023.

[68] 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.

[69] 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.

[70] O Ronneberger, P Fischer, and T Brox. Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015 Conference Proceedings*, 2022.

[71] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily L Denton, Kamyar Ghasemipour, Raphael Gontijo Lopes, Burcu Karagol Ayan, Tim Salimans, et al. Photorealistic text-to-image diffusion models with deep language understanding. *Advances in Neural Information Processing Systems*, 35:36479–36494, 2022.

[72] Tim Salimans and Jonathan Ho. Progressive distillation for fast sampling of diffusion models. In *International Conference on Learning Representations*, 2021.

[73] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.

[74] Nikhil Sheoran, Subrata Mitra, Vibhor Porwal, Siddharth Ghetia, Jatin Varshney, Tung Mai, Anup Rao, and Vikas Maddukuri. Conditional generative model based predicate-aware query approximation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8259–8266, 2022.

[75] Andy Shih, Suneel Belkhale, Stefano Ermon, Dorsa Sadigh, and Nima Anari. Parallel sampling of diffusion models, 2023.

[76] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. In *International Conference on Learning Representations*, 2020.

[77] Jianyi Wang, Kelvin CK Chan, and Chen Change Loy. Exploring clip for assessing the look and feel of images. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 2555–2563, 2023.

[78] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 233–248, 2023.

[79] Zijie J Wang, Evan Montoya, David Munechika, Haoyang Yang, Benjamin Hoover, and Duen Horng Chau. Diffusiondb: A large-scale prompt gallery dataset for text-to-image generative models. *arXiv preprint arXiv:2210.14896*, 2022.

[80] Zike Wu, Pan Zhou, Kenji Kawaguchi, and Hanwang Zhang. Fast diffusion model, 2023.

[81] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. Approximate query service on autonomous iot cameras. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 191–205, 2020.

[82] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. {VideoChef}: Efficient approximation for streaming video processing pipelines. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 43–56, 2018.

[83] Neeraja J Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 184–191, 2019.

[84] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518, 2021.

[85] Miao Yin, Yang Sui, Siyu Liao, and Bo Yuan. Towards efficient tensor decomposition-based dnn model compression with optimization framework. In *CVPR*, 2021.

[86] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[87] Chenshuang Zhang, Chaoning Zhang, Mengchun Zhang, and In So Kweon. Text-to-image diffusion model in generative ai: A survey. *arXiv preprint arXiv:2303.07909*, 2023.

[88] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, 2017.

[89] Yuxin Zhang, Weiming Dong, Fan Tang, Nisha Huang, Haibin Huang, Chongyang Ma, Tong-Yee Lee, Oliver Deussen, and Changsheng Xu. Prospect: Expanded conditioning for the personalization of attribute-aware image generation. *arXiv preprint arXiv:2305.16225*, 2023.

[90] Hongkai Zheng, Weili Nie, Arash Vahdat, Kamyar Azizzadenesheli, and Anima Anandkumar. Fast sampling of diffusion models via operator learning. In *International Conference on Machine Learning*, pages 42390–42402. PMLR, 2023.

A NIRVANA Algorithm

NIRVANA first uses the `match-predictor` to predict if there is a close match for the prompt query $\mathcal{P}_Q$. If the prediction is yes, it makes a `VDB` call to get the nearest prompt and finds the K using heuristics (see 5.2). Next, it retrieves the intermediate image noise at K^{th} step from `EFS` and passes it to the diffusion model to generate the final image in $N-K$ steps. If there is no match predicted or found, NIRVANA takes fallback to generate an image from scratch. Note, using `match-predictor` reduces the case where `VDB` call is made for *cache miss* cases.

Algorithm 2 GenerateImageCache($\mathcal{P}_Q$)

$P_e \leftarrow$ Embed(P)
if MatchPredictor(P_e) is $True$ **then**
  // Generate using cache
  (neigh, score) $\leftarrow$ search_VDB(P_e)
  $K \leftarrow$ heuristics_K(score)
  if $K \neq 0$ **then**
    $path_K \leftarrow$ neigh['payload']['noise'][K]
    $c\_noise \leftarrow$ retrieve_EFS($path_K$)
    $I \leftarrow$ model(P, c_noise, K)
  else
    // No suitable cache found, generate from scratch
    $I \leftarrow$ model(P, null, 0)
  end if
else
  // Generate from scratch
  $I \leftarrow$ model(P, null, 0)
end if
return I

B Implementation Details

Batch Processing: While deploying, our diffusion model takes up around 80% of GPU memory with batch size 1, so concurrent batching isn't supported.

AITemplate (AIT): We leverage AITemplate (AIT) [11], a Python framework to accelerate inference serving of the PyTorch-based diffusion model by converting it into CUDA (NVIDIA GPU) / HIP (AMD GPU) C++ code. It provides high-performance support during the inference process.

C Discussions

Image diversity. Since NIRVANA reuses previous intermediate states from the cache, over time the diversity of images generated by the system can be reduced if the majority of the prompts encountered by the system are very similar. This can be addressed in the following ways: first, by actively changing the `seed` of the denoising process after retrieval. We observed that for $K < 35$ this can increase diversity. However, if prompts become too similar, then NIRVANA would attempt to save compute more aggressively using a higher K where `seed` change becomes ineffective. In that case, NIRVANA can

be designed to let an ε fraction of prompts follow the vanilla diffusion process to maintain the diversity of images.

Change in prompt characteristics. If the characteristics of the prompts suddenly change due to some external factors, NIRVANA will find less similar items in the cache and will automatically move towards lower K, or even for more dissimilar prompts the `match-predictor` will kick in and redirect the prompts to vanilla diffusion process. While this will reduce compute savings, *by design* NIRVANA will not let the image quality degrade in case of a sudden change in prompt characteristics.

Generalizability of Approximate Caching: The concept of *approximate caching* easily extends to iterative generation approaches. It can also be applied to other architectures, such as Transformers (in LLMs) in sequential decoding [59], or at the layer level [56] for models like U-Net.

Cache management policy: The proposed `LCBFU` policy prioritizes compute savings, particularly with higher K values. However, there's a nuanced trade-off, notably for frequently accessed prompts at lower K (e.g., $K = 5$) v/s less frequent access at higher K (e.g., $K = 25$). Eviction at lower K may seem to impact the user experience by removing frequently used cache, but a lower $K = 5$ facilitates alternative matches with a lower similarity score requirement. Preserving $K = 25$ is critical for its similarity to specific prompts and substantial compute savings; evicting it might result in the absence of a highly similar prompt. Hence, (`LCBFU`) aims for maximum compute optimization with minimal impact on user experience.

ML optimizations. Several techniques are used to reduce the compute footprint and latency of models such as the use of lower precision [60], distillations [63, 72], pruning [38], batched-inference optimizations [22]. NIRVANA is complementary to such techniques as it can be used on top of those optimized models as well to reduce redundant computation using approximate caching. However, if a new family of generative models emerges that does not require such a large number of iterative steps, then NIRVANA's applicability will become limited. But since as of today, `DMs` with 50 or more denoising steps produce the best and production quality images, NIRVANA provides an attractive proposition for compute reduction.

D Additional Results from Sensitivity Analysis

D.1 Match-Predictor Settings

The *SGDOneClassSVM* `match-predictor` can produce binary predictions (0 or 1) by employing various thresholds. These thresholds influence the Precision (P) and Recall (R) values obtained from the `match-predictor`. To determine the optimal settings, we conducted an ablation study, measuring the overhead latency under different P and R configurations. The resulting plot in Figure 17 led us to select the settings with a $P = 96$ and $R = 95$. The choice is made to prioritize high precision, aiming to minimize false positives. Simultaneously, we aim to maintain a high recall to avoid missing opportunities for using *approximate-caching*.

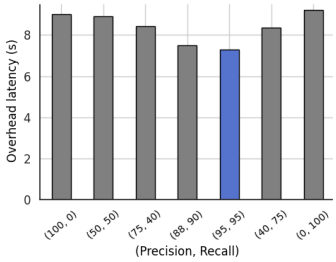


Figure 17: Hit rate and compute saved across K's

D.2 Decomposition of End-to-End Latency in NIRVANA

In Figure 18 we show the end-to-end latency of VANILLA diffusion model and also how different components of NIRVANA contribute towards its end-to-end latency. We can see noise retrieval from EFS and VDB search is the main contributor to the overhead.

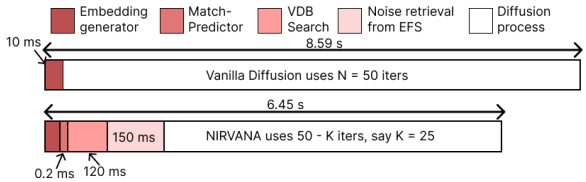


Figure 18: Time taken by different components of VANILLA and NIRVANA for generating an image using 50 steps on A10g GPU instance.

D.3 Image Quality across Long vs. Short Prompts

As we discussed in §3, the prompt queries can be either long or short. We perform ablation to see how the system works with them. The ablation results presented in Table 5 indicate that NIRVANA performs more effectively with slightly shorter prompts compared to very lengthy ones. This disparity in performance can be intuitively attributed to the retrieval technique employed. Longer prompts tend to challenge the ability of embeddings (in our case, CLIP) to capture the context adequately. If the prompt retrieved from the cache significantly deviates from the query despite high similarity based on *CLIPText* embeddings (as discussed in §3), it may result in the generation of incoherent images

Prompt	*FID* ↓	*CLIPScore* ↑
Short	7.96	28.48
Long	11.48	28.96

Table 5: Generation of short v/s long prompts. Less than 15 words are considered short.

D.4 Quality with different Caching Policy

We conducted an evaluation to compare image quality metrics, including *FID*, *CLIPScore*, and *PickScore*, while using different cache eviction techniques. The results, presented in Table 6, demonstrate that image generation quality remains

consistent across all caching techniques. Therefore, the choice of eviction mechanism should prioritize improved compute savings and hit rates.

Policy	*FID* ↓	*CLIPScore* ↑	*PickScore* ↑
FIFO	5.12	28.25	20.31
LRU	4.82	28.54	20.38
LFU	4.98	28.61	20.42
LCBFU	4.94	28.65	20.41

Table 6: Quality of generation for different eviction techniques with 1500 GB cache.

E Concept Development in Image Generation

This section gives a motivating example of how various concepts/characteristics develop during image generation [89]. In Figure 19, we present an illustrative example of image

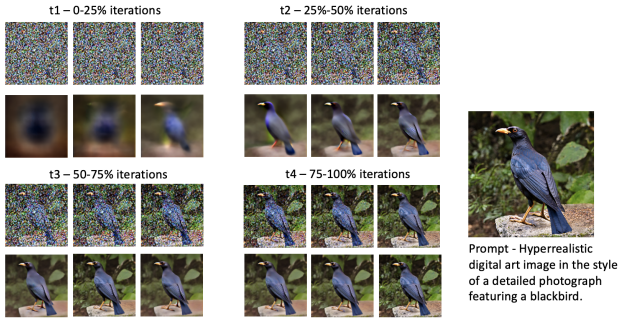


Figure 19: Generation across K

generation using a prompt that encompasses various aspects, including color, layout, content, size, style, and more. We divide the generation steps into four distinct time buckets, labeled t1, t2, t3, and t4, with each bucket having a unique role in shaping different facets of the image. The formation of color initiates in t1 and becomes relatively stable by t2. Style, here referring to generating a digital art image in the style of a detailed photograph, commences in t1 and experiences significant development in t2. The image's content, featuring a bird and leaves, begins to manifest partially towards the end of t2, with substantial development occurring in t3. Layout, which dictates the positioning of elements like the blackbird, starts its formation in t1 and progresses towards the end of this phase. By the time we reach t4, most aspects of the image are firmly established, with this final bucket primarily responsible for fine-tuning and enhancing details and clarity.