Department of Computer Science and Engineering

Indian Institute of Technology Bombay

Requests of a Feather Must Flock Together: Batch Size vs. Prefix Homogeneity in LLM Inference

CS496, Spring 2026

Saksham Rathi

Under Prof. Mythili Vutukuru

Contents

List of Tables

List of Figures

Abstract

Auto-regressive token generation in large language models is memory-bound because it requires "attending to" key and value tensors (KV cache) of all previous tokens. Prior work aims to improve the efficiency of this *decode* process by batching multiple requests together, and maximizing batch size subject to GPU memory constraints.

The key observation of our work is that, with prefix-sharing workloads, smaller, prefix-homogeneous batches – where *all* requests share a common prefix – can achieve higher decode throughput than larger, heterogeneous batches due to better spatial and temporal locality during KV cache accesses. However, prefix-aware schedulers in state-of-the-art inference engines maximize prefix reuse within a batch only to reduce the KV cache memory footprint but do not stop batch formation at smaller homogeneous batches that could have performed better. Furthermore, we show that shared prefix detection in existing schedulers relies on radix-tree traversals, incurring substantial CPU overhead that is often comparable to GPU execution time.

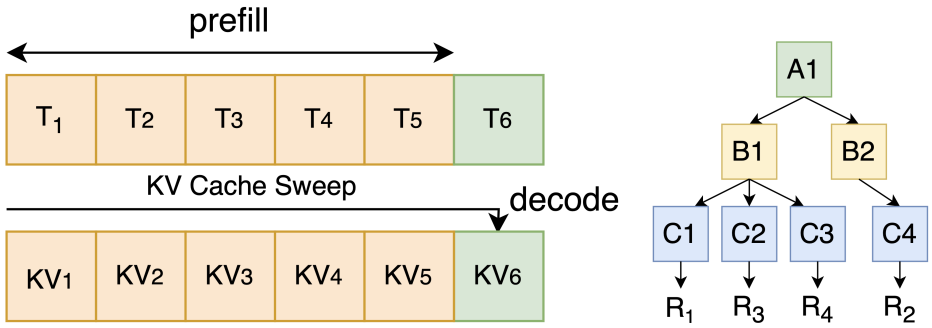
We present Feather, a prefix-aware scheduler that uses reinforcement learning (RL) to learn the optimal tradeoff between batch size and prefix homogeneity. We also introduce Chunked Hash Tree (CHT), a lightweight data structure that enables fast prefix detection and efficient request selection for the RL scheduler, avoiding expensive tree traversals. We integrate Feather into vLLM and SGLang, and our evaluation shows that Feather achieves 2–10× higher end-to-end throughput compared to existing schedulers, while performing no worse than the status quo when the workload does not have enough prefix sharing. Feather achieves these gains by reducing the total number of KV cache accesses, surpassing the performance of prefix-aware attention kernels that have the same goal.

1. Introduction

Large language models (LLMs) [1, 2, 3] are rapidly transforming modern computing, powering applications such as conversational assistants [4, 5], code generation [6], and retrieval-augmented generation [7, 8]. As these models transition from research to production, cloud AI engines [9, 10, 11] must improve serving latency and resource efficiency for both online and offline inference. While prior work has proposed scheduling strategies [12, 13, 14, 15], parallelization techniques [16, 17], memory management methods [18, 19, 20], quantization [21, 22], and kernel-level optimizations [23, 24], efficiently serving LLM workloads, especially under realistic high-throughput conditions, remains an open problem.

An LLM inference request goes through two phases: *prefill* and *decode*. The prefill phase processes all prompt tokens in parallel to produce the first output token, making it compute-bound. The decode phase, by contrast, generates tokens auto-regressively and is fundamentally memory-bound. Each newly generated token attends to the keys and values of all previous tokens, resulting in a sweep of the entire KV cache for each decode iteration, as shown in Figure 1.1(a). As context lengths grow, this memory traffic becomes the dominant bottleneck. Popular LLM inference engines like vLLM [18] adopt batching to improve GPU utilization during the decode phase, scheduling multiple requests together in a batch using policies like First Come First Serve (FCFS). More advanced frameworks explore techniques such as interleaving [15], disaggregation [13], and spatial co-scheduling [25, 26], along with dynamically adjusting batch sizes based on runtime constraints [12], to address this compute-memory mismatch.

Orthogonally, workloads such as few-shot prompting [27, 28], system prompts [29], and RAG [30] exhibit significant *prefix sharing* across requests. Reusing KV caches for shared prefixes eliminates redundant KV computation and reduces memory footprint. Systems such as SGLang [31] and vLLM [18] use radix trees (Figure 1.1(b)) or hashing-based structures to keep track of shared KV caches. While vLLM's FCFS would have scheduled requests in arrival order (R_1, R_2, R_3, R_4), SGLang performs a depth-first traversal of the radix tree and schedules requests in the order R_1, R_3, R_4, R_2, which reduces KV cache evictions. A complementary line of work [23, 24, 32, 33] exploits the shared prefix structure within the attention kernel itself by



(a) Prefill and Decode (b) Radix Tree

Figure 1.1: LLM Inference Primitives

jointly computing attention for groups of queries that share prefixes. Such prefix-aware attention kernels avoid redundant loading of shared KV blocks for each request separately, thereby reducing memory traffic.

Despite these advances, we identify two critical gaps in current scheduling strategies for prefix-shared workloads. First, prior work largely focuses on maximizing batch size during the decode phase. However, we show via controlled experiments that prefix homogeneity — the extent to which *all* requests in a batch share a common prefix — has an even stronger impact on performance. Further, we find that the performance gains increase with the length of the prefix shared across all the requests in a batch (§3.2). This is because when a prefix is shared by all requests, there is significant *spatial* and *temporal* locality in the memory accesses of the KV cache located in the GPU DRAM, which leads to both higher memory bandwidth utilization and reduced total amount of data fetched from DRAM. As a result, moderately small prefix-homogeneous batches can leverage these locality benefits and perform better than larger heterogeneous batches. For example, Figure 1.2 shows the throughput and execution times of 800 requests, where 8 groups of 100 requests each share a large prefix of 10K tokens between them. Processing these requests in 8 prefix-homogeneous batches of 100 requests each leads to nearly 2× higher throughput than when using larger heterogeneous batches, though very small homogeneous batches (32 batches of 25 requests each) perform poorly as well due to sub-optimal compute utilization. Existing prefix-aware schedulers have no mechanism to recognize this tradeoff between batch size and prefix homogeneity (§3.3); e.g., in Figure 1.1(b), a DFS-traversal-based scheduler will use R_2 to maximize batch size, even if a smaller batch (only R_1, R_3, R_4) could have performed better due to a longer common prefix. The second big gap with current prefix-aware schedulers is that they incur prohibitive CPU overheads in maintaining and traversing radix trees, which can account for 50–90% of total latency (§3.4).

We introduce FEATHER (§4), a prefix-aware scheduler that improves throughput by efficiently finding an optimal balance between batch size and prefix homogeneity. FEATHER is built on two components: *(i) Chunked Hash Tree (CHT)*, a lightweight data structure for fast prefix detection, similar in spirit to vLLM's block table [18]. Instead of performing token-level radix tree traversals, CHT represents each request as a vector of cumulative prefix hashes. To efficiently select a request that maximizes the length of the prefix shared across the active batch, CHT maintains a *min-heap*, updated lazily, and ordered by per-request *missing count*—the number of prefix chunks not yet covered by the batch. This enables the identification of the best candidate in $\mathcal{O}(\log W)$ time among W waiting requests, a significant improvement over DFS traversal [31].

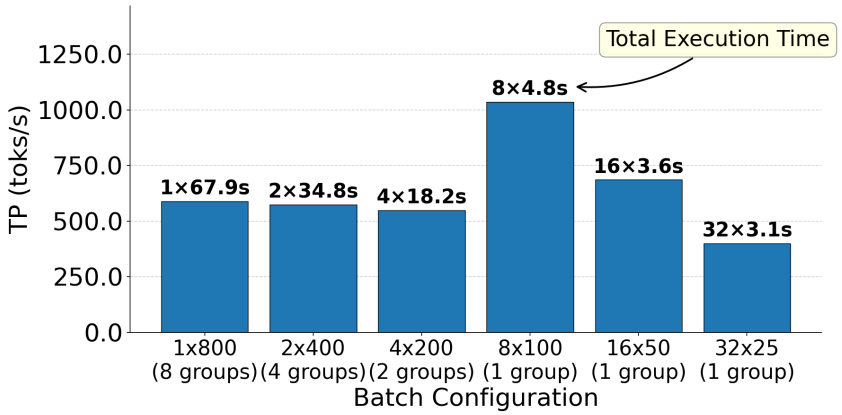


Figure 1.2: Batch Size vs. Prefix Homogeneity

(ii) RL-based batching policy. Deciding when to stop adding requests to a batch presents a trade-off: larger batches improve utilization but dilute prefix homogeneity. We cast this as a reinforcement learning problem in which the scheduler observes batch characteristics (e.g., current size, shared prefix length), evaluates the impact of adding another request to the batch, and stops batch formation at a point that maximizes throughput. This adaptive approach consistently outperforms static heuristics across diverse workloads and hardware configurations. Together, these two components enable Feather to construct prefix-homogeneous batches efficiently, both improving system throughput and reducing scheduling overhead.

We implement Feather on vLLM [18] and SGLang [31] and evaluate it across diverse workloads and configurations (§5). Our results show that Feather achieves 2-10× higher end-to-end throughput over vLLM's FCFS and other baselines for shared prefix lengths ranging from 1K to 10K tokens, while also improving DRAM bandwidth utilization. Feather is fully hardware-agnostic and yet performs better than recent prefix-aware kernels that must be tuned to specific hardware configurations. Our RL policy ensures that our performance never falls below that of the baselines, even if the workload does not allow for prefix-homogeneous batches, and CHT reduces prefix-detection overhead by up to 1000× compared to DFS-based approaches for long sequences. In summary, we make the following contributions:

- We demonstrate the tradeoff between batch size and prefix homogeneity in the decode phase performance of LLM inference and identify previously overlooked CPU overheads in prefix-aware scheduling.
- We design the Chunked Hash Tree (CHT) data structure for low-overhead prefix detection and develop a reinforcement learning-based policy for adaptive batch construction.
- We build Feather, a prefix-aware scheduler integrated into vLLM and SGLang, and demonstrate improvements in throughput and latency on diverse workloads. We plan to open-source Feather by contributing it to both frameworks' repositories.

1.1 Why the name Feather?

Feather is named after the structural pattern of prefix-homogeneous batches in our system: a shared prefix forms a central shaft, from which individual requests branch out as they diverge, much like the barbs of a Feather. The metaphor captures both the visual structure of our batches and the functional benefit of keeping prefix sharing requests together for better performance.

2. Background and Related Work

We begin with the fundamentals of language modeling and transformer architectures, then describe the inference process, optimizations in modern inference engines, and key performance metrics relevant to our study. We also discuss the prior work on batching, scheduling, and prefix-aware optimizations.

Language models (LMs) are statistical models that predict the next word in a sequence given the previous words. They basically provide us with a probability distribution over words based on the context given by the preceding words. They enable machines to understand and generate human language. They can be used for various tasks, such as answering questions, text summarization, translation, code generation, and more.

2.1 Stages in the Model Creation Pipeline

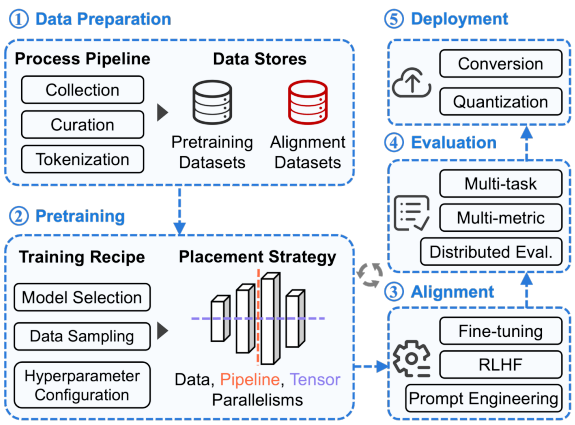


Figure 2.1: Stages in the Model Creation Pipeline [34]

- **Data Preparation:** The initial stage involves gathering and preprocessing the training data, which can be categorized into two parts: (1) pretraining data, consisting of extensive unlabeled corpora obtained from public or private sources and curated through processes like detoxification and deduplication; (2) alignment data, comprising a smaller set of high-quality labeled corpora used to align the model with specific tasks. This data is typically acquired through expensive human annotation or labeling. Besides, all the data must be tokenized to ensure compatibility with the model's input.
- **Pretraining:** It involves self-supervised training on large-scale curated data, demanding a majority of resources within the overall development workflow. Training LLMs efficiently at scale necessitates various system innovations, such as state-sharding optimizers and meticulous model placement using data, pipeline, and tensor parallelism.
- **Alignment:** This stage aims to adapt LLMs to user intent on a wide range of downstream tasks. Two primary aligning paradigms are commonly used: (1) prompt engineering, specifying prompts (i.e., inputs) without modifying model parameters. For example, in text summarization, appending a prompt "TL; DR" to the input article can improve model performance; (2) fine-tuning, updating model parameters on a task-specific dataset to improve performance in a particular domain.
- **Evaluation:** Given the vast application scenarios of LLM, it may be inaccurate to assess model quality solely based on a single metric like training loss. There are numerous factors to consider, such as accuracy, fairness, and toxicity. Consequently, it is crucial to account for a diverse set of criteria and measure performance across multiple tasks. Furthermore, regular evaluation is essential during the pretraining stage to provide timely feedback on model quality.
- **Deployment:** To meet the strict cost and latency constraints of LLM applications, several advanced techniques have been developed to achieve efficient model serving, including quantization, distillation, CUDA kernel optimization, model parallelism and memory management.

2.2 Transformer Architecture

2.2.1 Tokenization

Tokenization is the process of converting raw text into a sequence of tokens that the model can process. Instead of working directly with characters or words, modern language models often rely on subword tokenization methods such as Byte Pair Encoding (BPE), WordPiece, or SentencePiece. These approaches break words into smaller, more frequent units, balancing vocabulary size and representation power. For example, the word "unhappiness" may be split into "un", "happiness", while rare words are decomposed into character-level units. Tokenization ensures consistent representation of unseen or rare words and significantly reduces out-of-vocabulary issues.

2.2.2 Embedding and Attention

Once tokenized, tokens are mapped into continuous vector representations called embeddings. These embeddings capture semantic and syntactic relationships among words, such that similar words are close in the embedding space. Positional encodings are added to embeddings to preserve word order information, since attention mechanisms are permutation invariant by default.

Attention is the core mechanism of transformer models. It allows the model to selectively

focus on different parts of the input sequence when generating or predicting tokens. Given a query, attention computes weighted similarities with key-value pairs, enabling contextualized representations that consider both local and long-range dependencies.

2.2.3 Multi-Head Masked Attention

Masked attention is used in autoregressive language models to prevent the model from "looking ahead" at future tokens during training. This masking enforces causality by ensuring that each token can only attend to itself and preceding tokens. Multi-head attention extends this concept by computing multiple attention functions in parallel. Each head captures different aspects of relationships—such as syntactic dependencies, semantic similarity, or positional cues. The outputs of all heads are concatenated and linearly transformed, providing a richer, more expressive representation of the input.

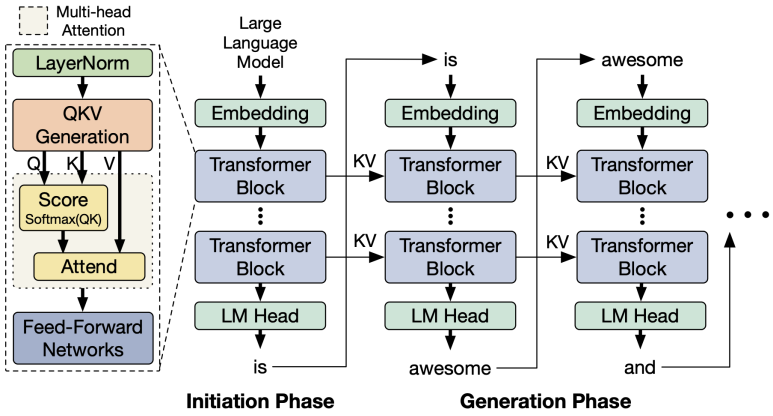


Figure 2.2: Architecture of a Large Language Model [35]

2.3 Transformer Based Large Langauge Models

The task of language modeling [18] is to model the probability of a list of tokens $(x_1, \ldots, x_n)$. Since language has a natural sequential ordering, it is common to factorize the joint probability over the whole sequence as the product of conditional probabilities (a.k.a. autoregressive decomposition):

$$P(x) = P(x_1) \cdot P(x_2|x_1) \ldots P(x_n|x_1, \ldots x_{n-1})$$

Transformers have become the de facto standard architecture for modeling the probability above at a large scale. The most important component of a Transformer-based language model is its self-attention layers. For an input hidden state sequence $(x_1, \ldots, x_n) \in \mathbb{R}^{n \times d}$, a self-attention layer first applies linear transformations on each position i to get the query, key, and value vectors:

$$q_i = W_Q x_i, \quad k_i = W_K x_i, \quad v_i = W_V x_i$$

Then, the self-attention layer computes the attention score a_{ij} by multiplying the query vector at one position with all the key vectors before it and compute the output o_i as the weighted

average over the value vectors:

$$a_{ij} = \frac{exp(q_i^T k_j/\sqrt{d})}{\sum_{t=1}^{i} exp(q_i^T k_t/\sqrt{d})}, \quad o_i = \sum_{j=1}^{i} a_{ij} v_j$$

Besides the above computation, all other components in the Transformer model, including the embedding layer, feed-forward layer, layer normalization, residual connection, output logit computation, and the query, key, and value transformation, are all applied independently position-wise in a form of $y_i = f(x_i)$.

- **Query (Q):** A representation of the current token, used to score its relevance against all past tokens. During decoding, we only compute the query for the token currently being processed.
- **Key (K):** Keys act like labels for tokens in the sequence. They are used to match queries and determine which past tokens are relevant.
- **Value (V):** Values are the actual token representations. Once relevance scores are computed (using Q and K), the corresponding values are aggregated to produce the output representation.

2.3.1 Autoregressive Generation

A request to an LLM service provides a list of input prompt tokens $(x_1, \ldots, x_n)$,and the LLM service generates a list of output tokens $(x_{n+1}, \ldots, x_{n+T})$. Due to the decomposition, the LLM can only sample and generate new tokens one by one, and the generation process of each new token depends on all the previous tokens in that sequence, specifically their key and value vectors. In this sequential generation process, the key and value vectors of existing tokens are often cached for generating future tokens, known as KV cache. Note that the KV cache of one token depends on all its previous tokens. This means that the KV cache of the same token appearing at different positions in a sequence will be different.

Given a request prompt, the generation computation in the LLM service can be decomposed into two phases:

The prompt phase takes the whole user prompt $(x_1, \ldots, x_n)$ as input and computes the probability of the first new token $P(x_{n+1}|x_1, \ldots, x_n)$. During this process, also generates the key vectors $k_1, \ldots, k_n$ and value vectors $v_1, \ldots, v_n$. Since prompt tokens $x_1, \ldots, x_n$ are all known, the computation of the prompt phase can be parallelized using matrix- matrix multiplication operations. Therefore, this phase can efficiently use the parallelism inherent in GPUs. This phase is also known as the **prefill** phase.

The autoregressive generation phase generates the remaining new tokens sequentially. At iteration t the model takes one token x_{n+t} as input and computes the probability $P(x_{n+t+1}|x_1, \ldots, x_{n+t})$ with the key vectors $k_1, \ldots, k_{n+t}$ and value vectors $v_1, \ldots, v_{n+t}$. Note that the key and value vectors at positions 1 to $n + t - 1$ are cached at previous iterations, only the new key and value vector k_{n+t} and v_{n+t} are computed at this iteration. This phase completes either when the sequence reaches a maximum length (specified by users or limited by LLMs) or when an end-of-sequence (<eos>) token is emitted. The computation at different iterations cannot be parallelized due to the data dependency and often uses matrix-vector multiplication, which is less efficient. As a result, this phase severely underutilizes GPU computation and becomes memory-bound, being responsible for most portion of the latency of a single request. This phase is also known as the **decode** phase.

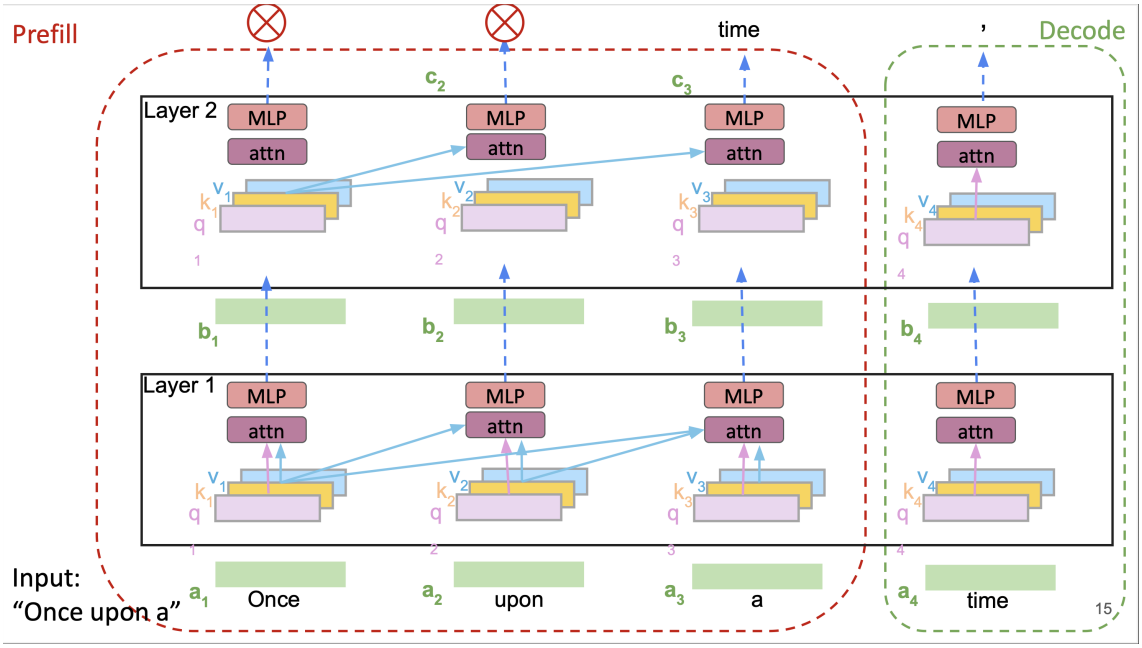


Figure 2.3: Prefill vs Decode [36]

2.4 LLM Inferencing

Large Language Model (LLM) inferencing refers to the process of using a trained model to generate predictions (such as text completions, answers, translations, etc.) given some input. Unlike training, where parameters are updated, inferencing only runs the trained model forward to produce outputs.

2.4.1 Forward Pass

- The forward pass is the core computation step in inferencing.
- Input tokens (converted to embeddings) are passed layer by layer through the transformer blocks:
  - Embedding → Attention → Feedforward → Normalization (repeated across layers).
- Each forward pass produces logits: a vector of unnormalized scores for the next token in the vocabulary.

2.4.2 Token Probability Distribution

- The logits are passed through a softmax to get probabilities.
- These probabilities define the model's belief about which token should come next.
- Different decoding strategies may be applied:
  - **Greedy decoding**: pick the token with maximum probability.
  - **Sampling**: sample a token according to the probability distribution.
  - **Top-k / Top-p sampling**: restrict sampling to the most probable tokens.
  - **Beam search**: keep multiple hypotheses alive and expand them.

2.4.3 Generation Loop

- Inferencing is auto-regressive:
  1. Generate one token.

2. Append it to the context.
3. Feed back into the model.

- Repeat until a stopping criterion (e.g., end-of-sequence token or maximum length) is reached.

2.4.4 Modes of Inference

- **Forward only** (logits for a given input): used in classification, scoring, and embeddings.
- **Generate** (auto-regressive token loop): used in chatbots, summarization, and translation.
- **Hybrid tasks**: e.g., retrieval-augmented generation (RAG), where inference integrates external knowledge.

2.5 Inference Engines

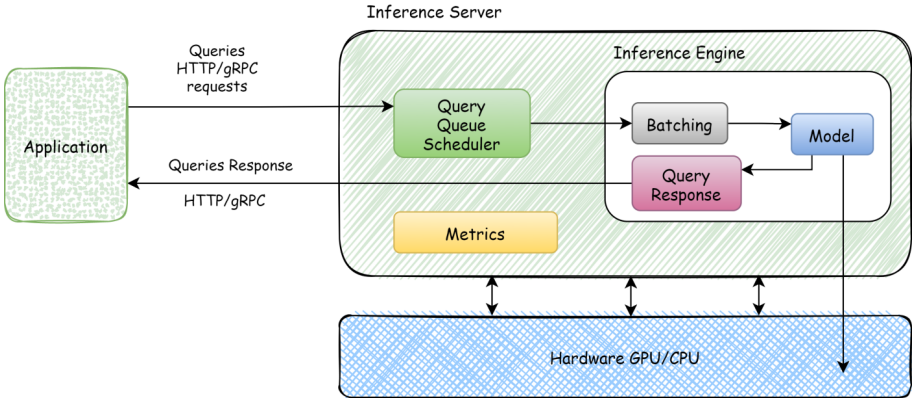


Figure 2.4: Architecture of an Inference Engine [35]

An **inference engine** is the software and systems layer that manages the execution of large language model (LLM) inference on hardware resources. While the model architecture and parameters define the computation, inference engines ensure that requests are executed efficiently, reliably, and at scale.

- **Batching:** Multiple user requests are grouped into a single batch to maximize GPU utilization and reduce per-request overhead. Efficient batching is crucial for achieving high throughput in production environments.
- **Streaming:** Outputs are generated token-by-token and streamed back to the user as they are produced, reducing latency for interactive applications such as chatbots.
- **Fault Management:** Inference engines must be resilient to hardware or software failures. This includes checkpointing, task retries, and robust scheduling.
- **Error Handling:** Engines provide mechanisms for handling issues such as invalid inputs, out-of-memory errors, or timeouts without disrupting other active requests.
- **Parallelism (Multi-GPU):** Large models often cannot fit on a single GPU. Inference engines handle model parallelism and pipeline parallelism to distribute computation across multiple GPUs or nodes, while minimizing communication overhead.

2.5.1 Optimizations in Inference Engines

- **Memory Storage Optimizations:** Inference engines carefully manage GPU memory by caching key-value pairs (KV cache), reusing memory buffers, and applying quantization or weight-sharing techniques to reduce footprint.
- **Offloading:** Portions of the model state or intermediate activations can be offloaded to CPU or other storage when not actively needed on the GPU. This allows inference of models larger than GPU memory capacity.

2.6 GPU Execution Model

GPUs are hierarchically organized for parallelism across hundreds of thousands of threads. The basic compute unit is the Streaming Multiprocessor (SM), and modern GPUs typically contain a few hundred SMs. Each SM includes an L1 cache, shared memory, tensor cores for matrix multiplication (GEMM), and execution units. SMs access global memory through a shared L2 cache. This hierarchy, from small, fast on-chip memory to large, high-latency global memory, is critical for LLM serving, as KV cache access patterns directly influence memory bandwidth utilization. Like CPUs, GPUs use hardware prefetching and benefit from regular, sequential memory access patterns [37].

2.7 Related Work

2.7.1 Batching and Scheduling

vLLM [18] adopts a first-come, first-served (FCFS) scheduling policy as a practical baseline for LLM serving. Sarathi-Serve [15] reduces prefill-decode interference through *chunked prefills* and stall-free batching, while DistServe [13] eliminates it by disaggregating the two phases across GPUs. At the kernel level, POD-Attention [25] overlaps prefill and decode on a single GPU via fused attention with SM-aware scheduling, and Bullet [26] enables fine-grained resource sharing through dynamic SM allocation. At the request level, BucketServe [14] groups requests by sequence length to improve batching efficiency, and **?** dynamically adapts the batch size to runtime constraints. Complementary to prior work, *Feather operates at the scheduler level and learns when to stop batch construction based on prefix homogeneity, introducing a scheduling signal not explicitly exploited in existing systems.*

2.7.2 Prefix Sharing

Many real-world LLM workloads [27, 28, 29, 30] exhibit substantial prefix overlap, and enabling KV cache reuse avoids redundant computation and reduces memory footprint. However, realizing these benefits requires coordination between serving and memory management. SGLang [31] formalizes prefix sharing via *RadixAttention*, which uses a radix tree to hierarchically organize token sequences, and enables KV reuse across shared prefixes. Similarly, vLLM [18] implements prefix caching [38] using a Merkle-tree-inspired hashing scheme (Figure 2.5), where each KV block hash depends on its tokens and preceding prefix hashes, enabling reuse when hash sequences match. BatchLLM [39] targets large offline workloads by grouping prefix-sharing requests and reordering them to maximize KV reuse. BlendServe [40] combines a resource-aware prefix tree with scheduling to jointly optimize prefix reuse and compute–memory demand. Feather adopts a lightweight design: *it uses chunked hashing over radix trees or full hash chains, reducing*

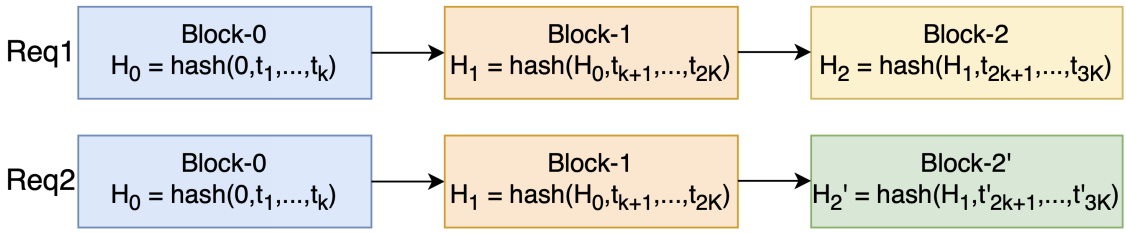


Figure 2.5: An Example of a vLLM Merkle Tree

CPU overhead and steering batch construction toward prefix-homogeneous groups in online and offline settings.

2.7.3 Prefix-Aware Attention Kernels

Prior work [23, 24, 32, 33] also optimizes attention kernels by grouping queries with shared prefixes to load KV blocks once and reuse them across queries. These approaches, however, rely on GPU-specific kernel designs. In contrast, *FEATHER improves inputs to the existing attention kernels by constructing prefix-homogeneous batches upfront, making it hardware-agnostic and naturally composable with kernel-level techniques.*

Next, we provide detailed studies of vLLM and SGLang, two state-of-the-art inference engines. We have implemented FEATHER on the top of both of these.

2.8 vLLM

vLLM [18] is an LLM serving system that achieves (1) near-zero waste in KV cache memory and (2) flexible sharing of KV cache within and across requests to further reduce memory usage. It also proposes PagedAttention, an attention algorithm inspired by classical virtual memory and paging techniques in operating systems.

Approximately 65% of the memory (for a 13B LLM model) is allocated for the model weights, which remain static during serving. Close to 30% of the memory is used to store the dynamic states of the requests. For Transformers, these states consist of the key and value tensors associated with the attention mechanism. The remaining small percentage of memory is used for other data, including activations - the ephemeral tensors created when performing inference on the LLM. Since the model weights are constant and the activations only occupy a small fraction of the GPU memory, the way the KV cache is managed is critical in determining the maximum batch size.

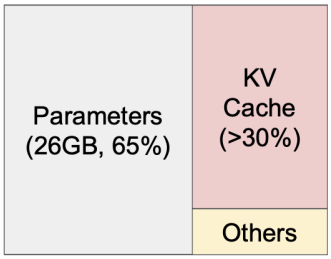


Figure 2.6: Memory layout when serving an LLM with 13B parameters on NVIDIA A100 [18]

Unlike the tensors in the traditional deep learning workloads, the KV cache has unique characteristics: it dynamically grows and shrinks over time as the model generates new tokens, and its lifetime and length are not known a priori. These characteristics make the existing systems' approach significantly inefficient in two ways: First, the existing systems suffer from internal and external memory fragmentation. To store the KV cache of a request in contiguous space, they pre-allocate a contiguous chunk of mem-

ory with the request's maximum length (e.g., 2048 tokens). This can result in severe internal fragmentation, since the request's actual length can be much shorter than its maximum length. Besides, external memory fragmentation can also be significant, since the preallocated size can be different for each request. Second, the existing systems cannot exploit the opportunities for memory sharing. LLM services often use advanced decoding algorithms, such as parallel sampling and beam search, that generate multiple outputs per request. In these scenarios, the request consists of multiple sequences that can partially share their KV cache. However, memory sharing is not possible in the existing systems because the KV cache of the sequences is stored in separate contiguous spaces.

2.8.1 Memory Challenges

- **Large KV Cache:** The KV Cache size grows quickly with the number of requests. As an example, for the 13B parameter OPT model, the KV cache of a single token demands 800KB of space, calculated as 2 (key and value vectors)× 5120 (hidden state size) × 40 (number of layers) × 2 (bytes per FP16). Since OPT can generate sequences up to 2048 tokens, the memory required to store the KV cache of one request can be as much as 1.6 GB.
- **Complex decoding algorithms:** LLM services offer a range of decoding algorithms for users to select from, each with varying implications for memory management complexity.
- **Scheduling for unknown input & output lengths:** The requests to an LLM service exhibit variability in their input and output lengths. This requires the memory management system to accommodate a wide range of prompt lengths. In addition, as the output length of a request grows at decoding, the memory required for its KV cache also expands and may exhaust available memory for incoming requests or ongoing generation for existing prompts.

2.8.2 vLLM

vLLM adopts a centralized scheduler to coordinate the execution of distributed GPU workers. The KV cache manager effectively manages the KV cache in a paged fashion, enabled by PagedAttention. Specifically, the KV cache manager manages the physical KV cache memory on the GPU workers through the instructions sent by the centralized scheduler.

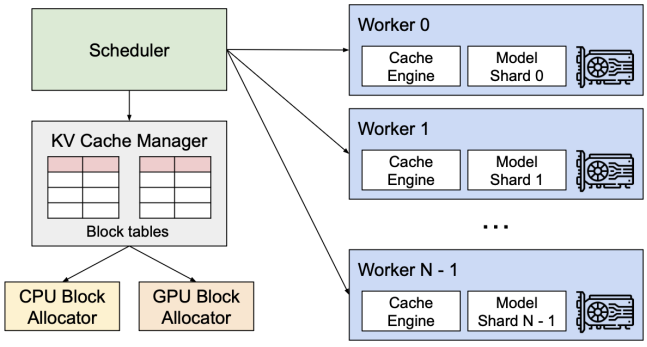


Figure 2.7: vLLM system overview [18]

PagedAttention

PagedAttention partitions the KV cache of each sequence into KV blocks. Each block contains the key and value vectors for a fixed number of tokens. During the attention computation, the PagedAttention kernel identifies and fetches different KV blocks separately. In summary, the PagedAttention algorithm allows the KV blocks to be stored in non-contiguous physical memory, which enables more flexible paged memory management in vLLM.

KV Cache Manager

A request's KV cache is represented as a series of logical KV blocks, filled from left to right as new tokens and their KV cache are generated. The last KV block's unfilled positions are reserved for future generations. On GPU workers, a block engine allocates a contiguous chunk of GPU DRAM and divides it into physical KV blocks (this is also done on CPU RAM for swapping). The KV block manager also maintains block tables—the mapping between logical and physical KV blocks of each request. Each block table entry records the corresponding physical blocks of a logical block and the number of filled positions. Separating logical and physical KV blocks allows vLLM to dynamically grow the KV cache memory without reserving it for all positions in advance, which eliminates most memory waste of the existing systems.

Parallel Sampling

In parallel sampling, one request includes multiple samples sharing the same input prompt, allowing the KV cache of the prompt to be shared as well. Since both outputs share the same prompt, space is reserved only for one copy of the prompt's state at the prompt phase; the logical blocks for the prompts of both sequences are mapped to the same physical blocks. Since a single physical block can be mapped to multiple logical blocks, a reference count is introduced for each physical block. At the generation phase, the two outputs sample different output tokens and need separate storage for KV cache. vLLM implements a copy-on-write mechanism at the block granularity for the physical blocks that need modification by multiple sequences. Unlike parallel decoding, beam search facilities sharing not only the initial prompt blocks but also other blocks across different candidates, and the sharing patterns dynamically change as the decoding process advances.

Scheduling and Preemption

Typically, eviction policies use heuristics to predict which block will be accessed furthest in the future and evict that block. Since in our case it's known that all blocks of a sequence are accessed together, we implement an all-or-nothing eviction policy, i.e., either evict all or none of the blocks of a sequence. Furthermore, multiple sequences within one request (e.g., beam candidates in one beam search request) are gang-scheduled as a sequence group. The sequences within one sequence group are always preempted or rescheduled together due to potential memory sharing across those sequences. There are two techniques to recover an evicted block:

- **Swapping:** In vLLM's case, evicted blocks are copied to the CPU memory. vLLM includes a CPU block allocator to manage the physical blocks swapped to CPU RAM. When vLLM exhausts free physical blocks for new tokens, it selects a set of sequences to evict and transfer their KV cache to the CPU. Once it preempts a sequence and evicts its blocks, vLLM stops accepting new requests until all preempted sequences are completed. Once a request completes, its blocks are freed from memory, and the blocks of a preempted sequence are brought back in to continue the processing of that sequence. Note that with this design, the number of blocks swapped to the CPU RAM never exceeds the number of total physical blocks in the GPU RAM, so the swap space on the CPU RAM is bounded by the GPU memory allocated for the KV cache.
- **Recomputation:** In this case, we simply recompute the KV cache when the preempted sequences are rescheduled. The performances of swapping and recomputation depend on the bandwidth between CPU RAM and GPU memory and the computation power of the GPU.

2.8.3 Automatic Prefix Caching in vLLM

Prefix caching [41] is a key optimization in LLM inference that avoids redundant computation by reusing previously computed `KVCacheBlocks`. When a new request shares a prefix with an earlier one, the system reuses the cached blocks instead of reprocessing the same tokens—resulting in large speedups without changing model outputs. This idea is widely adopted by production systems such as OpenAI and Anthropic, and implemented in open-source frameworks like vLLM and SGLang.

Overview

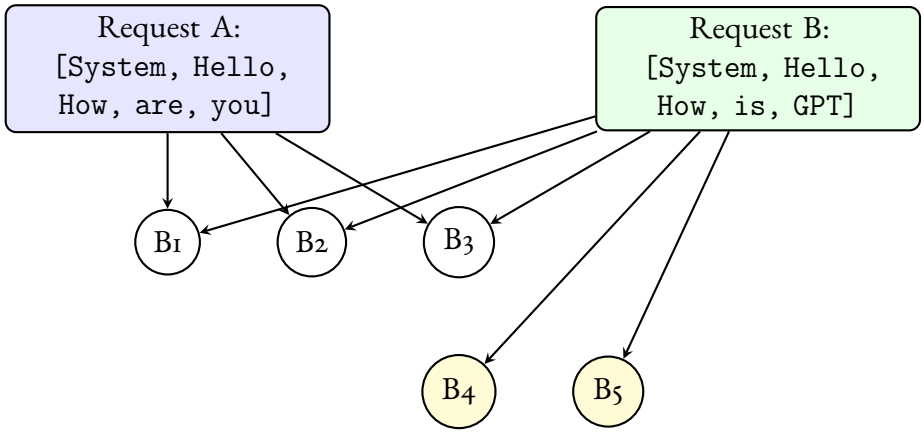


Figure 2.8: Illustration of prefix caching: shared prefixes reuse cached KV blocks.

Each cached block is identified by a unique hash computed from:

- **Parent hash:** hash of the preceding block.
- **Block tokens:** tuple of token IDs in this block (minimizes hash collisions).
- **Extra fields:** LoRA IDs, multi-modality hashes, or tenant-specific salts.

System Architecture

The KV cache manager maintains several internal structures:

- **Block Pool:** Pre-allocated list of `KVCacheBlock` instances to avoid dynamic object creation.
- **Free Block Queue:** A doubly linked list that tracks available blocks.
- **Cache Map:** Hash table mapping block hashes to block IDs.
- **Request Map:** Tracks which blocks belong to each active request.

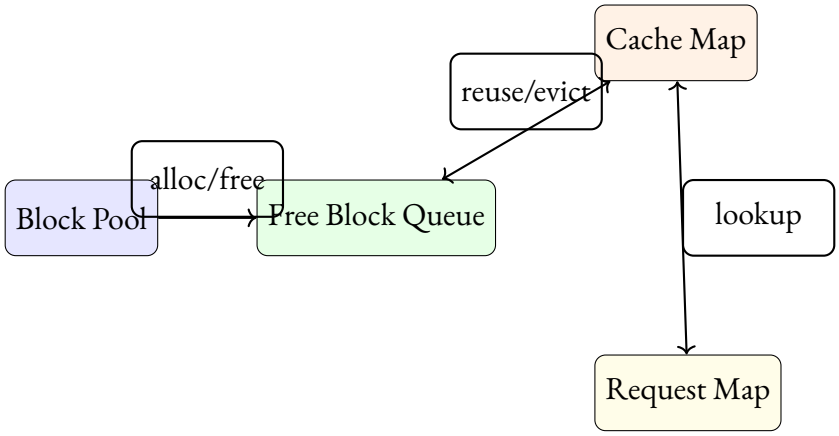


Figure 2.9: Internal organization of the KV cache manager.

Core Operations

1. Block Allocation

When the scheduler processes a new request:

1. It queries precomputed blocks via `get_computed_blocks()` using prefix hashes.
2. For each block:
    - Increment its reference count and remove it from the free queue.
    - Allocate new blocks as needed from the free queue head.
    - If allocation exceeds available capacity, evict the least recently used (LRU) blocks.

2. Free Operation

After a request completes:

- All blocks with zero references are appended to the tail of the free queue (in reverse order).
- Later blocks, which encode more tokens, are less likely to be reused soon.

3. Eviction Policy

When the head of the free queue holds a cached block:

1. Remove it from both the queue and cache map.
2. Delete the hash entry to maintain cache consistency.

2.9 SGLang

SGLANG [31] is a system for efficient execution of complex language model programs. SGLang consists of a frontend language and a runtime. The frontend simplifies programming with primitives for generation and parallelism control. The runtime accelerates execution with novel optimizations like RadixAttention for KV cache reuse and compressed finite state machines for faster structured output decoding.

2.9.1 LM Programs

The emergence of few-shot learning, self-consistency, skeleton-of-thought, tree-of-thought and various other patterns signifies a shift in our interaction with LLMs, moving from simple chatting to a more sophisticated form of programmatic usage of LLMs, which means using a program to schedule and control the generation processes of LLMs. These programs are referred to as LM (Language Model) programs. There are two common properties of LM programs:

- LM programs typically contain multiple LLM calls interspersed with control flow. This is needed to complete complex tasks and improve overall quality.
- LM programs receive structured inputs and produce structured outputs. This is needed to enable the composition of LM programs and to integrate LM programs into existing software systems.

2.9.2 Programming Model

Language primitives: SGLang provides primitives for controlling prompt state, generation, and parallelism. Here are the primitives: "gen" calls a model to generate and stores the results in a variable with the name specified in its first argument. It supports a "regex" argument to constrain the output to follow a grammar defined by a regular expression (e.g., a JSON schema). "select" calls a model to choose the highest probability option from a list. The operator "+=" or "extend" appends a string to the prompt. The operator "[variable_name]" fetches the results of a generation. "fork" creates parallel forks of the prompt state. "join" rejoins the prompt state. "image" and "video" take in image and video inputs.

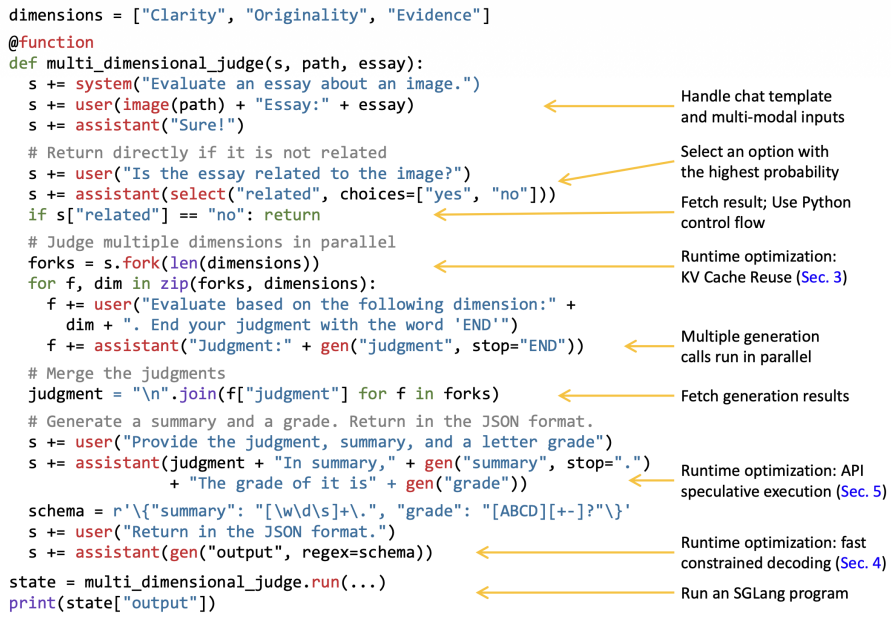


Figure 2.10: SGLANG Frontend Language Model [31]

2.9.3 RadixAttention

A radix tree is a data structure that serves as a space-efficient alternative to a classical trie (prefix tree). Unlike typical trees, the edges of a radix tree can be labeled not just with single elements but also with sequences of elements of varying lengths, significantly enhancing efficiency. In SGLANG, a radix tree is utilized to manage a mapping between sequences of tokens and their corresponding KV cache tensors. These KV cache tensors are stored in a non-contiguous, paged layout, where the size of each page is equivalent to one token. Because GPU memory is quickly filled by the KV cache, a simple LRU eviction policy is introduced that evicts the least recently used leaf first. By evicting leaves first, we enable the re-use of their common ancestors until those ancestors become leaves and are also evicted.

In the continuous batching setting, we cannot evict nodes used by the currently running

batch. Each node maintains a reference counter indicating how many running requests are using it. A node is evictable if its reference counter is zero. The cache hit rate is defined as $\frac{\texttt{number of cached prompt tokens}}{\texttt{number of prompt tokens}}$. If the request scheduler frequently switches between different, unrelated requests, it can lead to cache thrashing and a low hit rate. In the batch-processing setting, the requests are sorted by matched prefix length and prioritize requests with longer matched prefixes instead of using a first-come, first-served schedule.

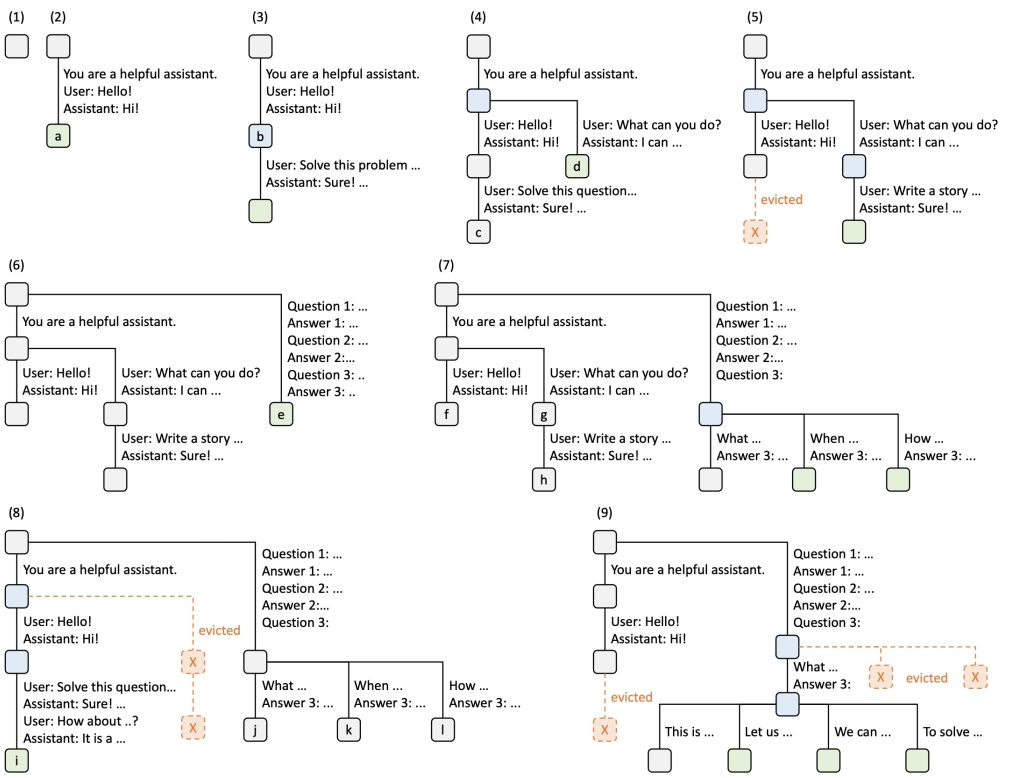


Figure 2.11: SGLANG Radix Tree Example [31]

2.9.4 Compressed Finite State Machine

In LM programs, users often want to constrain the model's output to follow specific formats, such as JSON schemas. Existing systems support this by converting a regular expression into a finite state machine (FSM). During decoding, they maintain the current FSM state, retrieve allowed tokens from the next states, and set the probability of invalid tokens to zero, decoding token by token. They can only decode one token at a time because the lack of integration between the FSM and the model runner in existing systems prevents multi-token processing, resulting in slow decoding. SGLang overcomes this limitation by creating a fast constrained decoding runtime with a compressed FSM. This runtime analyzes the FSM and compresses adjacent singular-transition edges in the FSM into single edges.

2.9.5 FastTree: Optimizing Radix Tree based KV Cache Computation

Existing systems still use conventional attention kernels that concatenate the KV cache per query and process each query independently. This leads to redundant memory loads of shared contexts from slow global memory and poor tensor core utilization due to mismatched tile dimensions. FastTree [24] introduces GPU kernels specifically designed to process queries that share common

prefixes within a radix tree structure. By adapting the computation to the tree topology at runtime, FastTree eliminates redundant memory operations and improves throughput.

Tree-structure Adaptive Runtime Optimization: Different ways of grouping queries—based on how prefixes are merged—can lead to drastically different computation patterns and memory-sharing behavior. Even with identical tree topologies, certain grouping plans might waste computation due to tile padding or cause additional global memory transactions when intermediate results must be written out. FastTree formulates this as an optimization problem: given a KV cache tree $T = (V, E)$, it searches for an edge assignment function $f : E \to \{0, 1\}$ that minimizes the end-to-end latency of the GPU attention kernel, balancing the trade-off between computation reuse and intermediate data costs. Padding overhead arises when the number of queries or context tokens fails to meet the tensor core tile size, wasting both shared memory and FLOPs. This is modeled through cost functions $C_{P,q}$ and $C_{P,c}$, which estimate padding-induced inefficiency at the query and context dimensions, respectively. Similarly, splitting the KV matrices introduces intermediate reduction overheads, modeled as SplitKV_Cost, capturing both padding and extra memory I/O costs. FastTree employs a greedy runtime algorithm (Fig. 2.13) that evaluates these costs per grouping plan and dynamically selects the most efficient configuration.

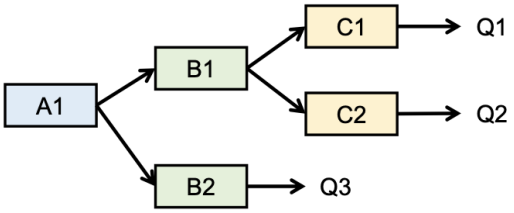


Figure 2.12: KV cache sharing through Radix Tree [24]

Efficient Attention Kernel Design: After determining the optimal grouping plan, FastTree performs the attention computation tile-by-tile within each group. It parallelizes the tiles of the query (Q) matrix across thread blocks while iterating over key (K) and value (V) tiles sequentially within each block due to inter-tile dependencies. By caching QKV tiles in shared memory and applying an online softmax, FastTree minimizes global memory transactions. Furthermore, aggregating multiple queries enables the transformation of GEMV into GEMM operations, thereby unlocking high tensor core utilization on modern GPUs.

Overall, FastTree demonstrates that by coupling radix tree–based KV cache organization with adaptive runtime planning and GPU-aware attention kernels, significant reductions in redundant computation and memory access can be achieved, leading to markedly higher throughput and lower latency for shared-context inference workloads.

Algorithm 1 Greedy Binary Edge Assignment

Input: Radix tree $T = (V, E)$

Initialize A as the number of all associated queries of each node in V
Initialize L as the context length of each node
for node v in BFS(V) **do**
  $nQ_{curr} = A[v]$ ▷ current #aggregated queries
  $len_v = L[v]$ ▷ accumulated context length
  for l in leaves(v) **do**
    $nQ_l = A[l]$, $len_l = L[l]$
    $C_0 = \text{SplitKVCost}(nQ_{curr}, nQ_l, len_l, len_v)$
    $C_1 = \text{SplitQCost}(nQ_{curr}, nQ_l, len_l, len_v)$
    if $C_0 \geq C_1$ **then**
      Assign 1 to edge $v \to l$
      $nQ_{curr} = nQ_{curr} - nQ_l$
      $L[l] = len_l + len_v$
    else
      Assign 0 to edge $v \to l$
    end if
  end for
end for

Figure 2.13: Greedy runtime optimization algorithm [24]

3. Motivation

Through a set of controlled experiments, we validate three hypotheses that underpin the design of Feather:

1. prefix homogeneity, the extent to which *all* requests in a batch share a common prefix, has a strong impact on decode throughput because it increases the locality of KV cache accesses. Importantly, these gains can be achieved *without any modifications to existing attention kernels*.
2. There is a trade-off between batch size and prefix homogeneity, where moderately sized homogeneous batches *can* outperform large heterogeneous batches; and
3. existing prefix detection approaches incur substantial CPU overhead.

3.1 Experimental Setup

All experiments use the Llama 3 8B [42] model and are run on an RTX 6000 Ada GPU [43] with 48 GB of GDDR6 memory and a 96 MB L2 cache. We use the vLLM [18] inference engine with a default maximum batch size of 500 unless stated otherwise. We first focus on decode throughput, measured in output tokens per second. To isolate decode behavior, we perform a warm-up run so that the prefix KV caches are resident in GPU memory. This ensures that per-token compute remains constant within an experiment and that differences arise primarily from memory accesses and batching. We also report end-to-end throughput, which includes the time taken to prefill the queries. For this model, a 10k-token request generates 1.2 GB of KV cache data across all layers. Within each layer, vLLM uses its paged memory management system and allocates the KV cache in blocks of 16 tokens contiguously in GPU memory, which is sufficient for locality benefits to kick in. We sample requests from the L-Eval [44] dataset and prune the size of each request as required by the experiment. We use the FlashAttention [45] backend throughout all the experiments.

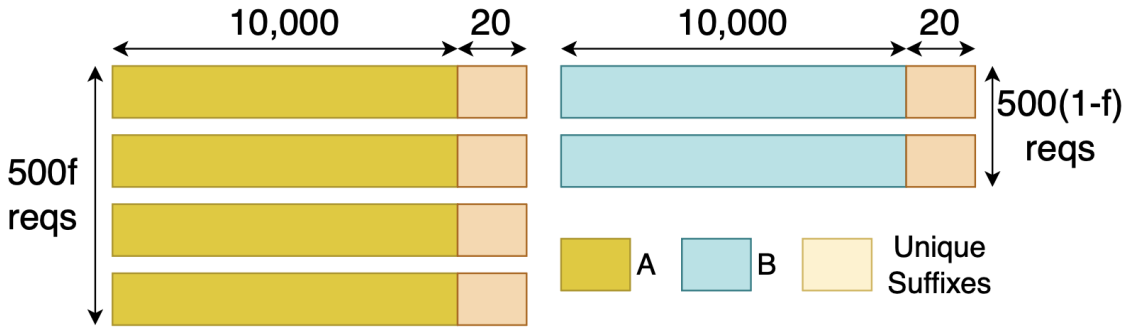


Figure 3.1: Two Large Prefixes

3.2 Significance of Prefix Homogeneity

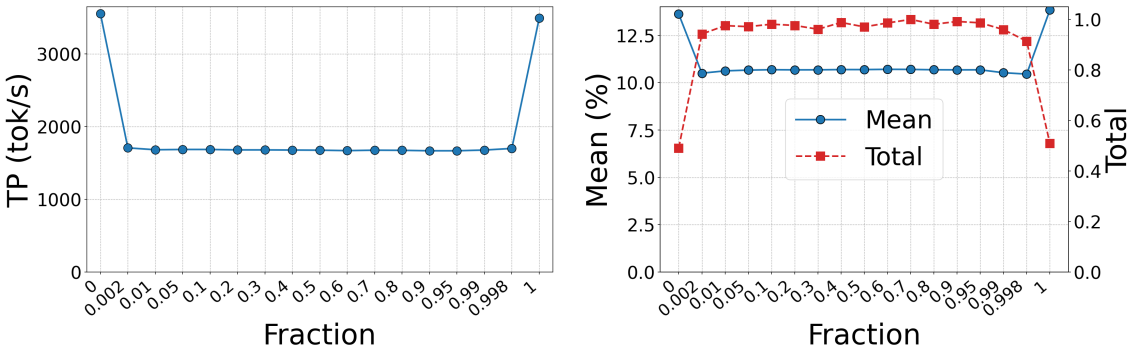
We start by observing the impact of prefix homogeneity on decode phase throughput. When *all* requests in a batch share a common prefix, their attention operations traverse the same KV cache regions concurrently. This yields two beneficial effects: (i) spatial locality: Sequential access to contiguous KV tensors enables the hardware prefetcher to fetch upcoming cache lines from DRAM into L2; (ii) temporal locality: Cache lines fetched for one request are reused by neighbouring requests before eviction. Together, these effects increase L2 hit rates and DRAM utilization, improving throughput. In contrast, heterogeneous prefixes interleave KV accesses across disjoint regions, disrupting locality and reducing effective memory bandwidth. We will now present experiments that validate these claims.

Experiment 1: We consider two prefixes, A and B, each 10K tokens long. We form a batch of 500 requests, each using the prefix A or B, followed by a unique suffix of length 20 tokens (Figure 3.1). Each request generates 50 decode tokens. Let $f \in [0, 1]$ denote the fraction of requests using prefix A. When f is 0 or 1, the batch is homogeneous; otherwise, two prefix groups coexist. We perform a warm-up run so that KV caches for both prefixes are resident in GPU memory.

Figure 3.2(a) shows decode throughput as a function of f. We see that the throughput is highest for fully homogeneous batches. However, even a single request from the other prefix group (e.g., $f = 0.002$) causes throughput to drop by nearly $2\times$. This demonstrates that decode throughput is highly sensitive to even minimal prefix heterogeneity. To understand this effect, we capture DRAM bandwidth utilization using DCGMI [46]. We first plot the mean bandwidth utilization for each f, and then we sum the utilization over the entire experiment duration (i.e., compute the area under the memory bandwidth curve) to approximate total memory accesses. As shown in Figure 3.2(b), homogeneous batches achieve over 40% higher bandwidth utilization than mixed-prefix batches due to greater spatial locality, enabling more efficient hardware-level prefetching into the GPU cache. Furthermore, greater temporal locality in prefix-homogeneous batches reduces the total data fetched from DRAM. Because most SMs process queries from the dominant prefix even when $f = 0.002$, we attribute the performance gains primarily to improved L2 cache reuse rather than increased L1 cache locality.

Takeaway 1: *Decode throughput is maximized when all requests share a prefix; even a single divergent prefix disrupts locality and sharply reduces effective bandwidth.*

Experiment 2: Next, we evaluate the impact of the shared prefix length on performance. For this, we construct a batch of 100 requests, each 2K tokens long and sharing a common prefix of length $2K \times f$ tokens, where $f \in [0, 1]$. The remaining $2K \times (1 - f)$ tokens are unique



(a) Throughput

(b) Bandwidth Utilization

Figure 3.2: Prefix Homogeneity - Two Prefix Groups

across requests (Figure 3.3). The KV caches fit into GPU memory for all values of f. Figure 3.4 shows that throughput increases smoothly with f. It is lowest at $f = 0$ and improves steadily as sharing increases, reaching over 60% higher throughput at $f = 1$. As f grows, a larger portion of the KV cache is shared across requests; since decode repeatedly traverses the KV cache, greater spatial and temporal locality improve throughput.

Takeaway 2: *In prefix-homogeneous batches, decode throughput increases smoothly with the length of the shared prefix.*

Experiment 3: Next, we evaluate the effect of the number of prefix groups on performance using an experiment with N shared prefixes $P_1, P_2, \ldots, P_N$, each of length $10K$ tokens and having same number of requests. Figure 3.5 shows throughput and average batch size as N increases. As observed earlier, throughput drops from $N = 1$ to $N = 2$ due to reduced temporal and spatial locality arising from prefix heterogeneity. However, from $N = 2$ to $N = 20$, throughput remains stable, indicating that additional heterogeneity has a limited impact once locality is disrupted. Finally, throughput drops sharply from $N = 20$ to $N = 100$, as GPU memory can no longer hold KV caches for all active prefixes. The impact is also visible through the reduced average batch size for $N = 50$ and $N = 100$. Prior work [31, 39] mitigates the throughput drop from $N = 20$ to $N = 50$ due to KV cache evictions by maximizing prefix reuse and minimizing memory footprint using depth-first traversal of radix trees. However, to the best of our knowledge, no prior work has observed the drop in performance from $N = 1$ to $N = 2$ prefix groups.

Takeaway 3: *While prefix heterogeneity causes a drop in throughput, the amount of heterogeneity has a marginal impact on the loss of locality. However, very high heterogeneity leads to minimal KV reuse and higher GPU memory evictions.*

Experiment 4: Until now, all the experiments have considered requests that have a single level of prefix sharing. We now ask the question: what is the impact of more complicated sharing patterns on performance? We consider four different radix tree sharing patterns. The sequence length in each case is fixed at $4L$. (i) In Figure 3.7(a), the entire $4L$ prefix is shared across all requests. (ii) In Figure 3.7(b), the tree has two levels. The first level, of length L, is shared by all requests, while the second level consists of two branches, each of length $3L$. (iii) In Figure 3.7(c), the tree has three levels with lengths L, L, and $2L$, and with 1, 2, and 4 branches at each level, respectively. (iv) In Figure 3.7(d), the tree has four levels, each of length L, with 1, 2, 4, and 8 branches at successive levels. Figure 3.6 shows the decode throughput for three values of L. As expected, a single level of radix tree has the highest performance because of maximum locality.

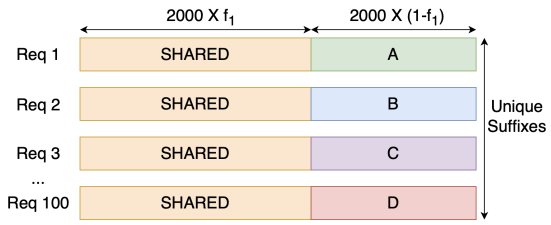


Figure 3.3: Fractional Sharing

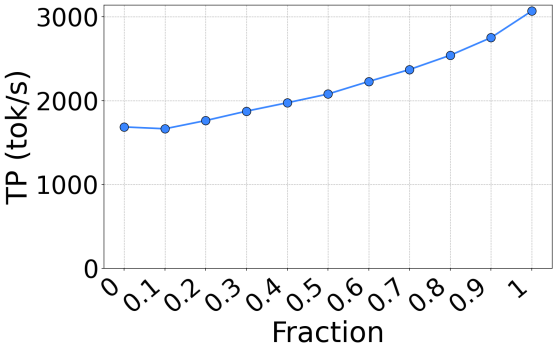


Figure 3.4: Fraction of Prefix Shared

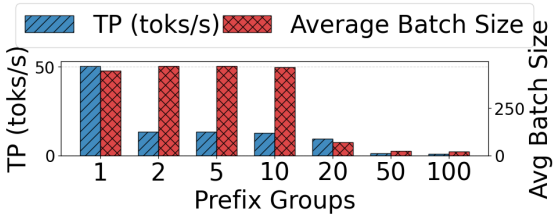


Figure 3.5: Number of Prefix Groups

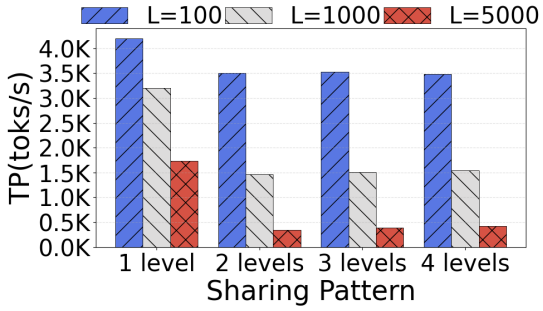


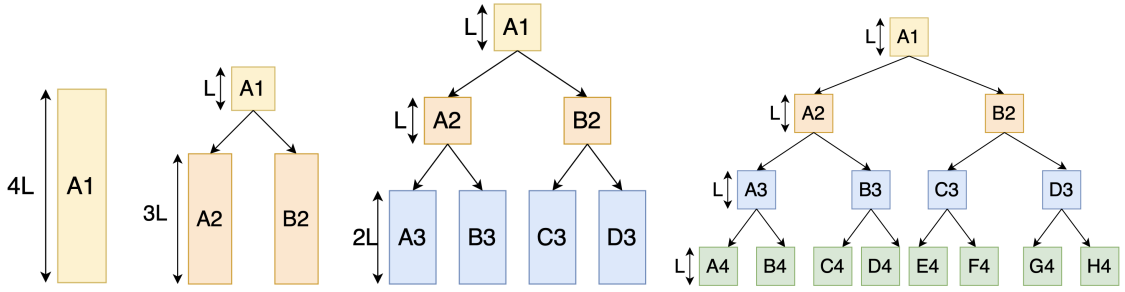
Figure 3.6: Radix Tree Sharing Patterns

Interestingly, radix trees with 2, 3, and 4 levels exhibit similar throughput across different values of L, despite differing branching structures. Why so? During the initial segment of length L, all configurations access the same KV region. However, once branching occurs, the batch becomes heterogeneous, locality degrades, and the marginal cost of additional prefix groups is small, as established earlier in Figure 3.5.

Takeaway 4: *Performance is driven mainly by the length of the top-level prefix shared by all requests, while the number of radix-tree levels and branches has a negligible impact.*

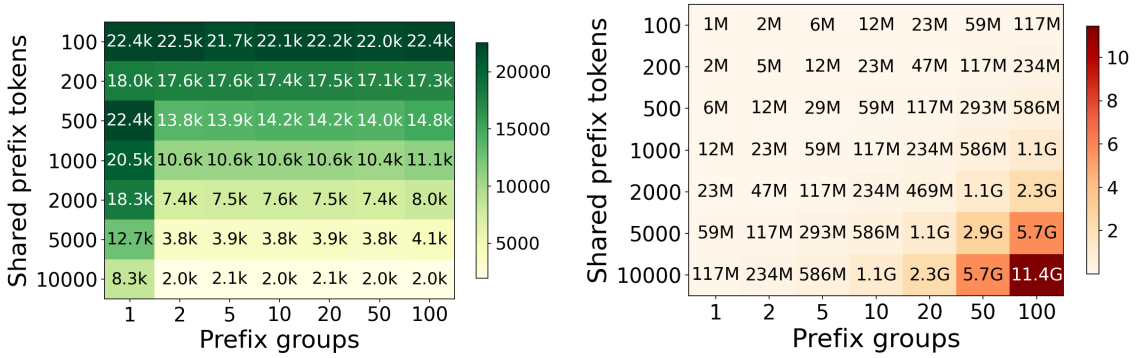
Experiment 5: We evaluate how total KV cache size and prefix group count jointly affect throughput using the Qwen 0.5B model, varying prefix lengths and numbers of prefix groups. Figure 3.8 shows vLLM's FCFS throughput alongside the total KV cache memory footprint for prefix tokens. As expected, throughput decreases as the prefix sequence length increases. A more pronounced drop is observed when the number of prefix groups increases from 1 to 2, which can be attributed to a loss of locality. Beyond this point, however, throughput remains relatively stable. This suggests that increasing the number of prefix groups further does not significantly degrade temporal or spatial locality, indicating that most of the locality loss occurs in the transition from a single group to multiple groups. Moreover, this drop from 1 to 2 prefix groups is observable only for larger shared prefix lengths where temporal and spatial locality benefits kick in.

Another question to ask if the high throughput values are correlated with the entire KV cache working set fitting into one of the cache levels. L1 cache size is typically in KBs, which is too small even for tens of tokens. Therefore, we focus on the L2 cache, which is 96 MB in size for the GPU on which these experiments were run. In the top-left region of the heatmap, the entire KV cache fits within this size. However, no clear throughput degradation is observed once this threshold is exceeded. This indicates that effective spatial and temporal locality depend



(a) 1-level (b) 2 levels (c) 3 levels (d) 4 levels

Figure 3.7: Radix Tree Sharing Levels



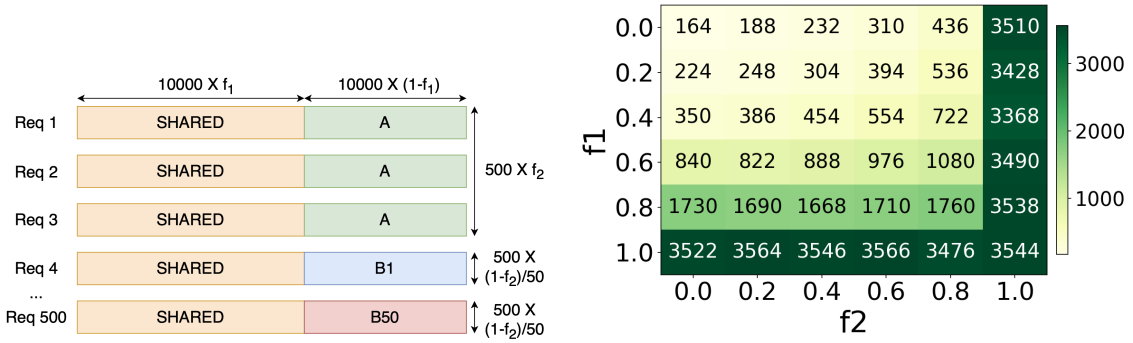
(a) Throughput (b) KV Cache Memory

Figure 3.8: Throughput and KV Cache Size

more on whether the subset of KV blocks actively accessed by batched requests at a given time (i.e. the dynamic working set) fits in the L2 cache. It is intuitive that this dynamic working set is smaller in size for prefix-homogeneous batches which explains the higher throughput gains achieved for such batches. Moreover, even for such a small model, the model weights are already large enough to prevent the entire KV cache to reside in L2. Therefore, performance gains due to prefix homogeneity primarily stem from a smaller dynamic working set of the KV cache.

Takeaway 5: *Throughput gains from prefix homogeneity stem primarily from a smaller dynamic working set of KV cache accesses, rather than from the total KV cache fitting within a particular cache level.*

Experiment 6: We consider an additional experiment where each request has a sequence length of $10K$ tokens. As illustrated in Figure 3.9(a), a fraction $f_1 \in [0, 1]$ of this sequence is shared as a common prefix across all 500 requests in the batch. The remaining $10K(1 - f_1)$ tokens constitute the suffix of each request. Within the batch, a fraction $f_2 \in [0, 1]$ of requests share an identical suffix, denoted as A. The remaining fraction $(1 - f_2)$ has suffixes drawn from one of 50 distinct groups, $B_1, \ldots, B_{50}$. Figure 3.9(b) presents the throughput as we vary f_1 and f_2. Consistent with the previous observations, increasing f_1 improves throughput due to enhanced temporal and spatial locality in KV cache accesses. Additionally, we observe a sharp increase in throughput when f_2 changes from 0.8 to 1. This occurs because at $f_2 = 1$, the entire batch becomes homogeneous, resulting in fully aligned KV cache traversal. Throughput is significantly lower at smaller values of f_1, primarily because limited GPU memory leads to frequent cache evictions. This is further supported by the increase in throughput with rising f_2



(a) Visualization (b) Throughput

Figure 3.9: Key Observations on Prefix Homogeneity

at low f_1, as more requests share the suffix A, reducing eviction frequency. However, once the working set fits within GPU memory (around $f_1 = 0.8$), throughput remains nearly constant for all $f_2 < 1$, indicating that residual heterogeneity has minimal impact. Overall, this experiment consolidates the previous key insights.

Takeaway 6: *Full batch homogeneity yields a disproportionate throughput gain over partial homogeneity. Moreover, increasing the fraction of prefix shared also leads to higher performance.*

3.3 Batch Size vs Prefix Homogeneity

So far, we have established that prefix homogeneity has a significant impact on performance. But when the workload does not have enough requests from a prefix group and prefix-homogeneous batches are too small, the compute will remain under-utilized, leading to a loss in throughput. Therefore, there exists a trade-off between batch size and prefix homogeneity. We now explore this trade-off.

Experiment 7: We study decode throughput vs. batch size using two workloads: (i) heterogeneous, with requests spread across 5 distinct 10K-token prefixes, and (ii) homogeneous, with all requests sharing one such prefix. Figure 3.10 shows throughput and tensor core utilization as a function of batch size for both workloads. We see that throughput plateaus at a point before the maximum batch size, and the saturation throughput is higher in the homogeneous case due to greater locality. Further, the batch size at which throughput saturates is higher for the homogeneous workload because requests can access the same KV cache without using additional memory bandwidth. Interestingly, the compute utilization is almost exactly the same across both workloads.

Takeaway 7: *Decode is memory bandwidth bound and increasing batch size beyond a point does not necessarily improve throughput. However, the point at which diminishing returns begin depends on prefix homogeneity (and hardware config).*

Experiment 8: Given that batch size has a diminishing impact on performance, we ask if moderately small homogeneous batches can outperform larger heterogeneous ones. To answer this, we use eight prefix groups with 100 requests each (800 total, each 10K tokens) and evaluate the following batching strategies: (i) 1×800 (8 prefix groups/batch); (ii) 2×400 (4 prefix groups/batch); (iii) 4×200 (2 prefix groups/ batch); (iv) 8×100 (homogeneous); (v) 16×50 (homogeneous); and (vi) 32×25 (homogeneous). We vary the decode length from 1 to 100 tokens and measure the end-to-end throughput, including both the prefill and decode phases for all

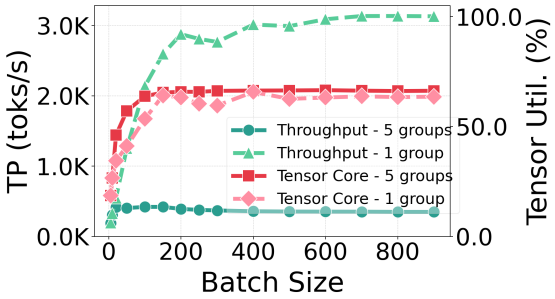


Figure 3.10: Throughput vs. Batch Size

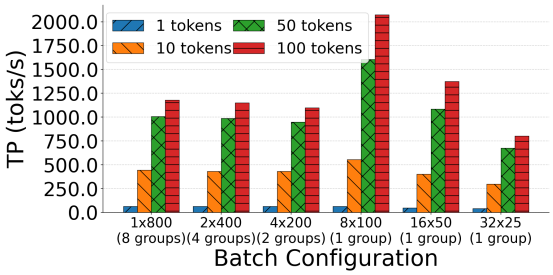


Figure 3.11: Batch Size vs. Homogeneity

batches. Figure 3.11 shows this throughput across the different batching strategies. For small decode lengths, throughput is dominated by prefill, and large heterogeneous batches perform the best due to higher tensor core utilization; splitting 1×800 into smaller batches slightly reduces throughput due to lower compute efficiency, as locality benefits are not yet dominant. As decode length increases, performance trends reverse: multiple decode iterations repeatedly sweep the KV cache and reap greater benefits due to locality. Therefore, at longer decode lengths, fully homogeneous batches (8×100) significantly improve throughput despite smaller batch sizes. Notably, even 16×50 continues to outperform 1×800 at large decode lengths, showing that locality gains can outweigh reduced tensor core utilization. Only when batches become too small (32×25) does throughput decline again, as reduced arithmetic intensity and occupancy dominate.

Takeaway 8: *Moderately small prefix-homogeneous batches can achieve higher throughput than larger heterogeneous ones. Gains increase with longer decode lengths as repeated KV sweeps amplify locality; however, excessively small batches underutilize compute and degrade performance.*

Policy	Exec (s)	Batch Time (s, %)	Throughput (toks/s)
vLLM FCFS	229.36	3e-6 (0.00%)	436.25
SGLANG DFS-W	177.78	92.15 (51.8%)	562.61
SGLANG LPM	149.27	46.62 (31.2%)	669.92
PAT (vLLM Block Table)	316.45	91.72 (29.0%)	316.63
vLLM RadixTree	139.33	62.95 (45.2%)	717.35
vLLM User IDs	105.43	0.03 (0.03%)	948.54

Table 3.1: Overhead of Dynamic Prefix Detection

3.4 Overhead of Dynamic Prefix Detection

Next, we study how state-of-the-art inference engines identify prefix sharing and the costs involved. Systems such as SGLang [31] and BatchLLM [39] do not explicitly form prefix-homogeneous batches, but use radix trees to detect sharing, and schedule requests to minimize KV recomputation.

Experiment 9: We construct a workload of 2K requests, each with a 10K-token prefix, and compare scheduling CPU time to total execution time. SGLang proposes two prefix-aware policies that use its radix tree: *Longest Prefix Match*, which prioritizes requests with the longest

matching cached prefixes, and *DFS-Weight*, which performs a depth-first traversal and prioritizes branches with more active requests. vLLM uses a much simpler hash-based block table, which is used for grouping requests in the prefix-aware attention kernel implementation of PAT [23]. We compare the overheads of this block table along with our own radix-tree-based scheduler implemented on top of vLLM. We also compare against an idealized policy where prefix group identifiers are provided explicitly, eliminating dynamic prefix discovery.

Table 3.1 shows that SGLang's LPM and DFS-Weight incur scheduler overhead (measured as time taken by all functions invoked during batch formation) comparable to GPU execution time, even though these policies improve performance over vLLM's FCFS. In contrast, when prefix group identifiers are provided explicitly, CPU overhead drops significantly. This gap arises from the structure of radix trees: they require token-by-token comparisons during insertion, dynamic node splitting on partial matches, and repeated tree traversals for scheduling requests. With long prompts and high concurrency, these operations incur significant overhead. Similarly, vLLM's block table design is inefficient in this regard. However, these overheads are not sufficiently highlighted in prior work [39, 40], which is mostly based on offline inference.

Takeaway 9: *Existing approaches introduce significant CPU overhead when detecting prefix sharing, which is sometimes even comparable to the GPU execution time.*

4. Design and Implementation

Our experiments so far have demonstrated that the length of the prefix shared across *all* active requests is an important determinant of system performance. However, this metric interacts with batch size in a complex way: larger batches improve GPU utilization but often reduce shared prefix alignment. Furthermore, detecting this shared prefix length incurs high overhead in existing systems. We now describe the design of FEATHER, an efficient scheduler that finds the right balance between batch size and prefix homogeneity.

4.1 Overview

FEATHER is based on the following two key ideas.

4.1.1 Key Idea 1: Learn the Stopping Decision via RL (§4.3)

The trade-off between batch size and prefix homogeneity raises a key question: when should the scheduler stop adding requests? The answer depends on both hardware and workload, making fixed heuristics brittle. We model batch construction as a sequential decision process and learn a stopping policy via Reinforcement Learning. At each step, the scheduler evaluates a candidate request using various state features and decides whether it should be added to the batch or if the batch should be dispatched to the GPU for execution.

4.1.2 Key Idea 2: Shared Prefix Detection via Chunked Hash Tree (§4.2)

Tree-based approaches incur significant overhead (§3.4), and finding the best waiting request requires a DFS traversal that is $\mathcal{O}(E)$, where E is the number of edges in the radix tree. In the worst case, this can go up to $\mathcal{O}(W \times T)$, where W is the number of waiting requests, and T is the sequence length. We instead represent each request as a sequence of hashes over fixed-size chunks and maintain a working set of hashes for the active batch. This is similar in spirit to Merkle-style hash chains, but it does not require sequential hash computation, and each chunk can be processed independently without prefix dependencies. Prefix matching is thus reduced

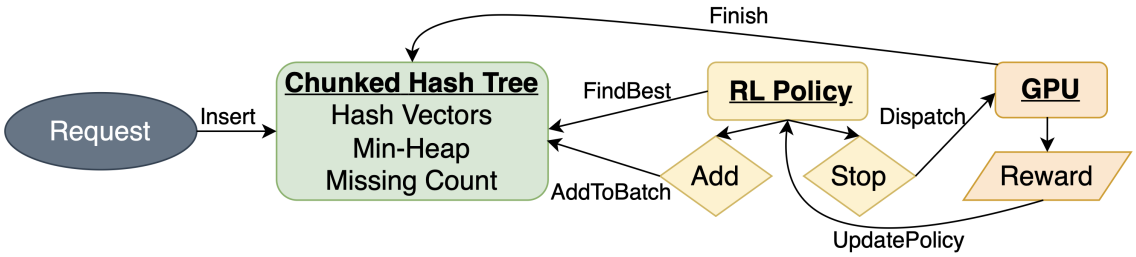


Figure 4.1: FEATHER Pipeline

to a set-overlap problem: selecting the request with the maximum overlap with the working set. However, a naive implementation that scans all requests and counts overlaps requires $\mathcal{O}(W \cdot C)$ work per decision, where C is the number of chunks in a request, which remains too expensive for frequent scheduling. We avoid this by maintaining a missing count per request—the number of chunks absent from the working set—which changes only when the working set changes (i.e., upon request addition or completion). Overlaps are thus updated incrementally, reducing selection to finding the request with the smallest missing count. To do so efficiently, we maintain a min-heap over missing counts, pushing updates and using lazy deletion to discard stale entries, yielding $\mathcal{O}(\log W)$ selection cost.

4.1.3 System Architecture

Figure 4.1 shows the overall scheduling pipeline. We consider an inference server that continuously receives requests, which are inserted into the waiting queue of CHT. The scheduler builds batches incrementally in a loop. It first finds the best candidate by retrieving the waiting request with the highest overlap with the current working set. Then the RL policy evaluates the current system state and decides whether to ADD the request or STOP. If ADD is chosen, the request is added to the scheduler batch, and all the data structures, such as the working set and the min-heap, are updated accordingly; otherwise, the batch is dispatched to the GPU for an execution step. The decode throughput measured after the GPU execution is used as a reward for the RL policy. Completed requests are removed from the active batch, and the working set is updated.

4.2 Chunked Hash Tree (CHT)

We now describe the operations of CHT (a more formal treatment is in §A). We begin with how token sequences are mapped to a compact representation in our system.

4.2.1 Hash Vectors

Given a request as a token sequence, we partition it into non-overlapping chunks of size K and build a sequence of hashes, one for each chunk, where each hash is cumulative over all tokens up to that chunk. For example, in Figure 4.2(a), a request with tokens A-F and $K = 2$ is represented as three incremental hashes: hash(A-B), hash(A-D), and hash(A-F). Two requests will have the same hash at level l iff they share the same first lK tokens. Each request is represented as an ordered hash vector. We use an incremental xxHash-64 state, ensuring each token is processed once, yielding a total cost of $\mathcal{O}(T)$, independent of the number of chunk boundaries, where T is the sequence length.

4.2.2 Insertion

INSERT is invoked when a new request arrives at the scheduler. We first compute the cumulative prefix-hash vector and store it in a map from request IDs to hash vectors for faster reuse. We also maintain a reverse index that maps each hash to the set of requests containing that hash. For each hash in the hash vector, we insert the request into this reverse index. The overall cost is dominated by the hash computation phase, which runs in $\mathcal{O}(T)$.

4.2.3 Working Set, Min-Heap and Shared Prefix Tip

To evaluate how well a request matches the active batch, we maintain a working set that contains (level, hash) tuples representing prefix hashes at that level covered by currently active requests. For each request, we also maintain a missing count, which is the number of levels for which this request's hashes are not present in the working set. We insert these missing counts into a min-heap, which enables efficient retrieval of the most suitable request for addition to the batch. INSERT updates both of these structures. Moreover, we do not perform in-place updates and use lazy deletion to discard stale entries in the heap. To relate to our key metric, which is the number of chunks shared by all active requests, we also maintain a shared prefix tip, storing the depth and hash of the deepest node that is shared by all the active requests. Figure 4.2(b) shows the Chunked Hash Tree after the request R_1 is added to the batch. The nodes marked in green denote the working set. Since there is just a single request, the shared prefix tip is at Level-3 of R_1. The heap contains the missing counts of the waiting requests along with stale entries.

4.2.4 Finding the Best Request

The goal of FINDBEST is to select the waiting request that best complements the active batch, i.e., the one with the minimum missing count, so that we maximize the depth of the shared prefix tip[1]. Since the min-heap is keyed by this count, the best candidate appears at the top after skipping stale entries. As shown in Figure 4.2(c), R_2 appears at the top of the min-heap and is therefore selected as the best request. Note that adding R_2 to the batch reduces the shared prefix tip by one level. FINDBEST runs in $\mathcal{O}(\log W)$, where W is the number of waiting requests. For the selected request, we compute two metrics that are inputs to the RL policy (§4.3), which needs to decide whether adding the request to the current batch is beneficial for the overall throughput. One of the metrics is the change in the depth of the shared prefix tip, and the other is the number of additional waiting requests that are present at this new depth, which can all be added to the batch without any further impact on the shared prefix tip in the future.

4.2.5 Adding a Request to the Active Batch

When a request is promoted to the active batch, we update the working set to include its prefixes and recompute the shared prefix tip. We iterate over the hash vector backwards; for each level, if the corresponding prefix hash isn't present in the working set, it is inserted, and all waiting requests sharing it have their missing count decremented. For the shared prefix tip, admitting a new request can only *shorten* the common prefix, so we seek the deepest level where it agrees with the existing tip hash. This corresponds to the least common ancestor of all active request leaves in the hash trie. ADDTOBATCH runs in $\mathcal{O}(C \cdot W_h \cdot \log W)$ time, where $C = \lceil T/K \rceil$ is

[1] An astute reader might ask why we are choosing a request based on its difference from the working set rather than its impact on the length of the shared prefix tip. A brief explanation is that we want to pick a request that helps us revert back to a longer shared prefix tip sooner, and a request that differs from the working set by the least number of chunks helps us achieve this goal best. We refer the reader to Appendix A.9 for more details.

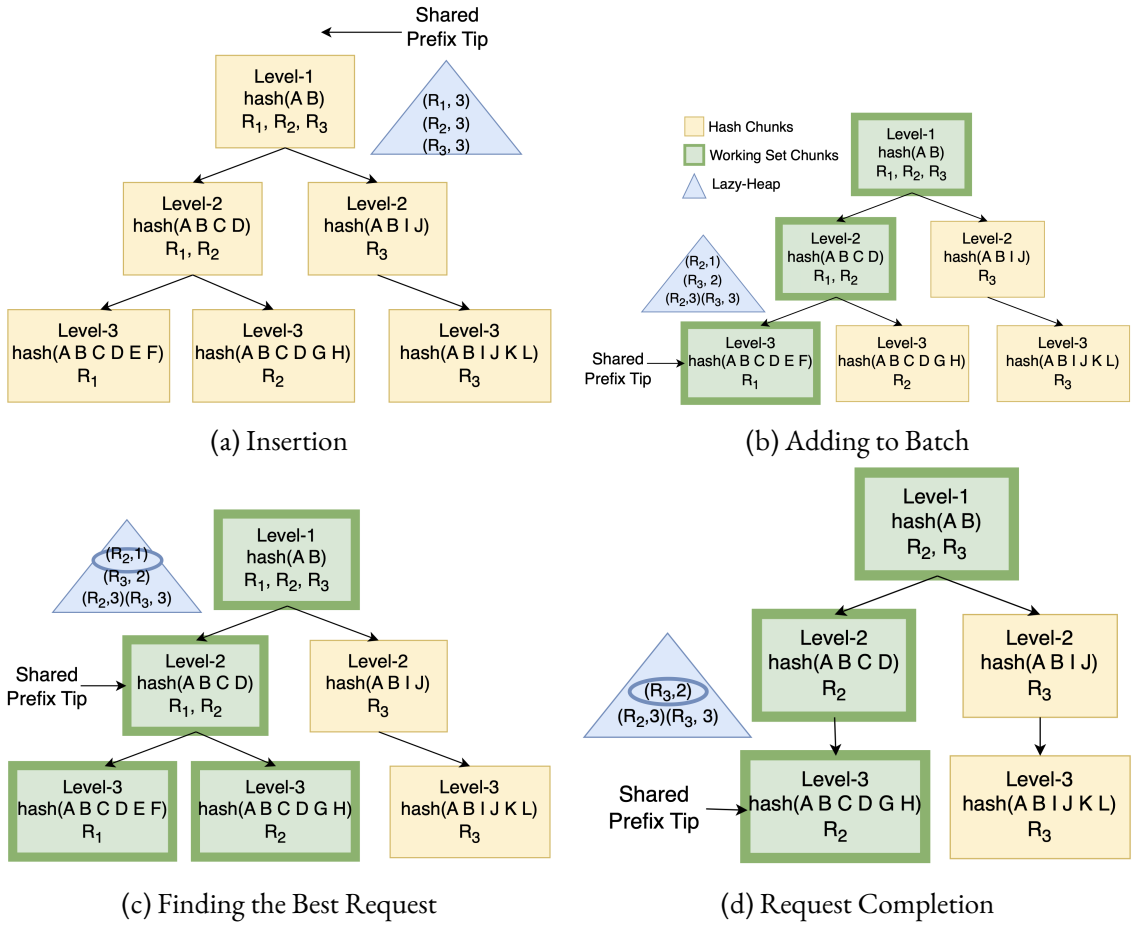


Figure 4.2: Chunked Hash Tree Operations

the number of chunks for a request with T tokens, and W_h is the number of waiting requests that share the same hash ($W_h \ll W$).

4.2.6 Removing a Request from the Active Batch

FINISH is called when a request completes, and we remove it from the active batch. It is the symmetric counterpart of ADDTOBATCH. We iterate over the hash vector from the start. A hash unused by any active request is removed from the working set, and all waiting requests containing it have their missing count incremented. A request completion can only *lengthen* the shared prefix tip since the remaining active requests may agree on a longer common prefix. Starting from the tip's previous position, we scan forward, checking whether all remaining active requests share the same hash at each level. (We do this efficiently by maintaining reference counts of each hash.) The extension stops at the first level of disagreement or at the shortest active request. This scan is inexpensive in practice, as the tip rarely advances more than a few levels. FINISH also runs in $\mathcal{O}(C \cdot W_h \cdot \log W)$ time. Figure 4.2(d) shows the state after R_1 completes: the tip extends to R_2's final level, and waiting heap costs are updated accordingly.

4.2.7 Optimizations

We employ several optimizations to improve efficiency. First, FINDBEST may be invoked multiple times within a scheduling round without state changes, so we cache its last result and reuse it

until it is invalidated by any mutation (e.g., AddToBatch and Finish). Second, we use lazy heap updates: when a request's missing count changes, we push a new entry instead of updating it in place and skip stale entries during FindBest. Third, we maintain the shared prefix tip incrementally rather than recomputing it. This drops the cost from $\mathcal{O}(A \cdot C)$, where A is the number of active requests, and C is the number of chunks, to just $\mathcal{O}(C)$ in the worst case and typically terminates earlier. Finally, all major data structures are pre-allocated at initialization to eliminate rehashing and reallocation overhead.

4.3 Batching and Reinforcement Learning

The CHT efficiently identifies the next best request to add, maximizing shared prefix length, but it does not determine *when to stop*. While adding requests improves GPU utilization, it can also shrink the shared prefix and degrade locality. We frame this stop/continue decision as a learning problem and present three policies: a heuristic, a contextual bandit, and a Q-learning agent. The scheduler incrementally builds a batch by selecting candidates via the CHT and deciding whether to admit them or dispatch the batch.

4.3.1 State, Actions, and Reward

At each step of batch formation, the scheduler observes the state $s = (b,\ \Delta,\ w)$ and takes an action $a \in \{\text{Add}, \text{Stop}\}$. Here, b is the current batch size (a proxy for GPU utilization), Δ is the loss in the shared prefix, and w is the number of waiting requests sharing the prefix up to the new depth, indicating whether the locality cost of this addition will be compensated by future activations. Both RL policies optimize the reward $\mathcal{R} = \text{throughput}_{\text{decode}}$. Maximizing this encourages large batches and short decode times (better prefix locality). The state is discretized into $(\hat{b}, \hat{\Delta}, \hat{w})$, where b and w use exponential bins, and Δ uses coarse thresholds. The resulting table is small, hashable, and converges quickly under moderate load.

4.3.2 Heuristic Policy

As a baseline, we encode domain intuition as simple rules balancing two goals: maximizing GPU utilization (large batches) and preserving prefix locality (avoiding large drops in the shared chain). The heuristic always accepts candidates with zero chunk loss (free additions). When the batch is small, it tolerates higher loss to avoid underutilization. Otherwise, it admits a candidate only if the chunk loss is within a fixed threshold, with slightly relaxed tolerance when many waiting requests share the resulting prefix (anticipating amortization). If none of these conditions hold, it stops and dispatches the batch. This policy requires no training and converges immediately, but its fixed thresholds cannot adapt to workload or hardware changes.

4.3.3 Contextual Bandit

To avoid hand-tuned thresholds, we use a contextual bandit with Upper Confidence Bound (UCB) selection [47]. Each batch-construction step is treated as an independent decision: given the discretized state $\hat{s} = (\hat{b}, \hat{\Delta}, \hat{w})$, the policy selects the action that maximizes

$$\text{UCB}(\hat{s}, a) = \frac{\mu(\hat{s}, a)}{n(\hat{s}, a)} + c\sqrt{\frac{\ln S}{n(\hat{s}, a)}}.$$

Here, $\mu(\hat{s}, a)$ is the cumulative reward, $n(\hat{s}, a)$ is the number of times action a was taken in $\hat{s}$, S is the total number of decisions, and c is the exploration coefficient. The first term favors high-reward actions, while the second promotes exploration of under-sampled ones.

4.3.4 Q-Learning Policy

As an alternative to the previous policy, we can also model batch construction as an episodic reinforcement learning problem, where each episode begins when forming a new batch and ends upon STOP or when the waiting queue is exhausted. The Q-table [48] stores values $Q(s, a)$ for each (discretized state, action) pair and is updated after each episode using:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big].$$

Here, r is the observed throughput, s' is the next state, α the learning rate, and γ the discount factor, which biases the agent toward near-term throughput, consistent with real-time inference. We follow an ε-greedy exploration strategy: at each step, a random action is selected with probability ε, and the greedy action $\arg\max_a Q(s, a)$ otherwise. During training, ε decays from 1.0 to $\varepsilon_{\min}$, transitioning from full exploration to near-deterministic behavior. Qualitatively, we expect this policy to perform better in the long term when actions influence future batch composition.

At each step, FINDBEST returns the lowest-cost candidate along with (Δ, w), forming the state s. The policy (heuristic, bandit, or Q-learning) then selects ADD or STOP. If ADD, the request is activated and appended to the batch; otherwise, the batch is dispatched. After execution, the observed decode throughput is used as the reward: the bandit updates its statistics and the Q-learning agent applies the Bellman update. This has also been shown in Figure 4.1.

4.4 Integration with vLLM and SGLang

We integrate our scheduler as a lightweight layer on top of both vLLM [18] and SGLang [31]. The Chunked Hash Tree (implemented in C++) along with shared prefix chain tracking and reinforcement learning policies operates entirely at the scheduler level. We will upstream our changes to these repositories upon acceptance of our publication.

5. Evaluation

We evaluate Feather across diverse models, GPU configurations, and real-world datasets and scenarios. Our evaluation answers the following questions:

- Under what workloads do smaller homogeneous batches outperform larger heterogeneous ones?
- How effective is Feather's Chunked Hash Tree compared to traditional radix-tree-based approaches?
- How do different design components of Feather (RL, CHT) interact to produce end-to-end performance gains?

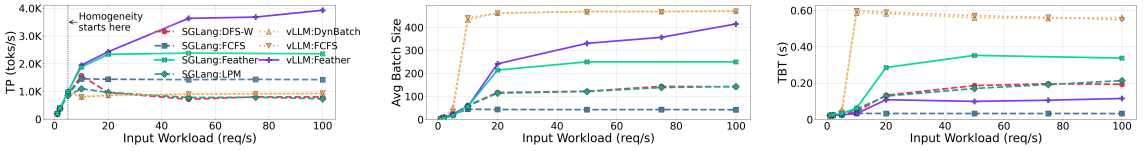
5.1 Experimental Setup

5.1.1 Models and Hardware

We evaluate three model families across multiple sizes: Qwen 0.5B, 1.5B, 8B [49], LLaMA 3 8B [42], and LongChat 13B [50]. Experiments are run on an NVIDIA RTX 6000 Ada GPU [43] with 48 GB GDDR6, 96 MB L2, and the maximum batch size is fixed at 500, which fits within the GPU memory for most settings. We additionally evaluate on an A100-80GB GPU for comparisons involving PAT [23], as it is specifically optimized for this hardware.

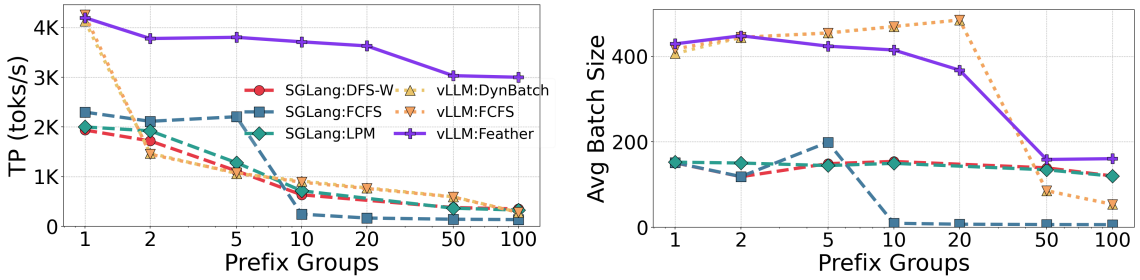
5.1.2 Workloads

Following prior work, request arrivals are modeled as a Poisson process, with length pruning applied as required by each experiment. L-Eval [44] is a benchmark of human-labeled query–response pairs spanning tasks such as summarization and question answering, with sequence lengths ranging from 2.7K to 210.5K tokens. LongBench [51] is a collection of long-context samples across six task categories: multi-document QA, summarization, and code completion, with context lengths between 4K and 10K tokens.



(a) Throughput (b) Average Batch Size (c) Time between tokens (TBT)

Figure 5.1: Poisson Workload



(a) Throughput (b) Average Batch Size

Figure 5.2: Varying Number of Prefix Groups

5.1.3 Baselines

We compare FEATHER against state-of-the-art LLM serving systems, all with prefix caching enabled. vLLM [18] uses its default FCFS scheduling policy. SGLang [31] is evaluated under FCFS, LPM, and DFS-W to cover its design space. We also evaluate DynamicBatching [12] and PAT [23], both implemented on top of vLLM: DynamicBatching adjusts batch sizes based on runtime memory utilization and latency constraints, while PAT is a prefix-aware attention kernel that reduces redundant KV cache accesses via query packing.

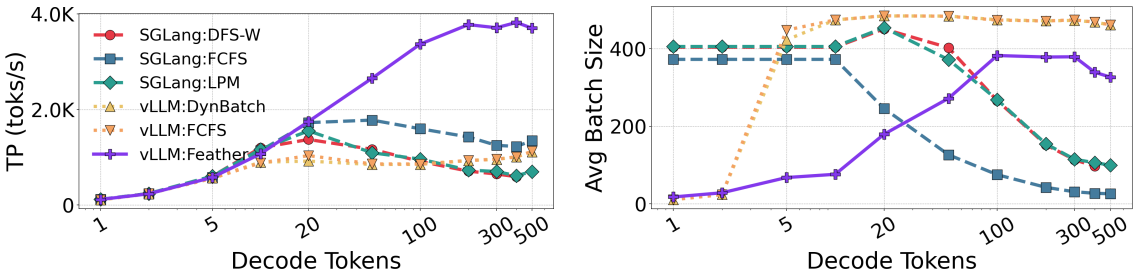
5.1.4 Metrics

We report three key metrics. Throughput (toks/s) measures the output tokens generated per second, reflecting serving capacity and GPU utilization. Time between tokens (TBT) measures the average interval between decoded tokens, reflecting responsiveness, especially in interactive settings. We also track the average batch size per forward pass to relate scheduling decisions to these metrics.

5.2 End-to-End Comparison: FEATHER vs Baselines

We evaluate a workload where the default configuration has 5 prefix groups[1], each 5K tokens long, with roughly equal request distribution, a request rate of 100 req/s, and 200 output tokens per request. Such workload configurations are common in prior work [23, 24] evaluating LLM serving systems. We vary these parameters to examine how FEATHER compares to the baselines. The token budget is 32K tokens. For FEATHER's RL policy, we find that bandit and Q-learning approaches perform similarly across most settings; unless stated otherwise, we report results using the bandit policy.

[1]We also evaluate workloads without prefix sharing and show FEATHER performs on par with FCFS (§5.3.7).



(a) Throughput (b) Average Batch Size

Figure 5.3: Varying Number of Decode Tokens

5.2.1 Varying Input Request Rate

Figure 5.1(a) shows throughput (toks/s) as a function of the request rate. Across all policies, throughput increases with the request rate before saturating. FEATHER (on vLLM) consistently outperforms all baselines at every request rate. At 100 req/s, FEATHER achieves roughly $4\times$ the throughput of vLLM (FCFS). Notably, this advantage is not driven by larger batch sizes. As shown in Figure 5.1(b), FEATHER in general maintains a *smaller* average batch size than vLLM (FCFS), and beyond the vertical line shown in the plot, FEATHER's RL policy decides to have completely homogeneous batches. This reinforces a key insight: *smaller, homogeneous batches outperform larger, heterogeneous ones*. The gradual growth in FEATHER's batch size with increasing request rate reflects its ability to efficiently aggregate more requests from the same prefix group into the active batch without sacrificing locality. SGLang policies consistently operate at smaller batch sizes than vLLM, as vLLM avoids double-counting shared prefix tokens toward the token budget, allowing more requests per batch. Dynamic Batching performs similarly to vLLM (FCFS), as all KV caches fit within GPU memory. Figure 5.1(c) shows time between tokens (TBT). FEATHER (on vLLM) achieves substantially lower TBT than all baselines except SGLang (FCFS), whose smaller batch sizes yield faster per-token generation. While SGLang's LPM and DFS-W policies also use small batch sizes, their TBT remains significantly higher than FEATHER due to non-trivial scheduling overhead during batch construction. FEATHER implemented on SGLang also shows similar qualitative trends and outperforms all other SGLang policies. Unless otherwise stated, all subsequent results refer to FEATHER implemented on vLLM.

5.2.2 Varying Number of Prefix Groups

Figures 5.2(a) and 5.2(b) show throughput and average batch size as functions of the number of prefix groups. Increasing the number of prefix groups introduces greater request heterogeneity, reducing KV cache locality. This leads to a sharp throughput drop for vLLM (FCFS), even when increasing from 1 to 2 prefix groups. As the number of groups increases further, FCFS degrades rapidly: beyond 20 groups, the KV cache footprint exceeds HBM capacity, causing a sharp reduction in batch size and increased recomputation. FEATHER mitigates this by grouping requests with shared prefixes, increasing KV cache reuse and avoiding premature evictions. As a result, FEATHER maintains a more stable batch size and throughput across all prefix group configurations, significantly outperforming all baselines. At 100 prefix groups, FEATHER achieves $10\times$ higher throughput than vLLM (FCFS). SGLang's LPM and DFS-W policies partially mitigate evictions, achieving higher throughput than FCFS (SGLang) at larger group counts.

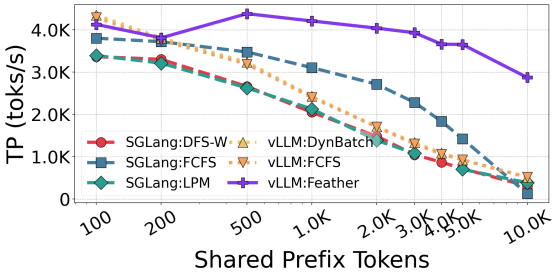


Figure 5.4: Varying Shared Prefix Tokens

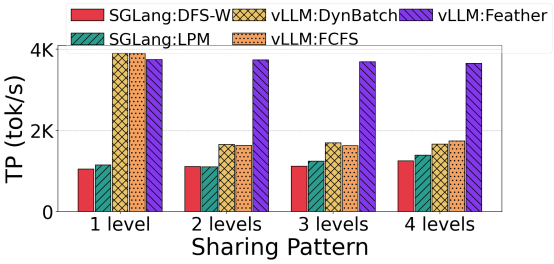


Figure 5.5: Varying Radix Tree Sharing Levels

5.2.3 Varying Number of Decode Tokens

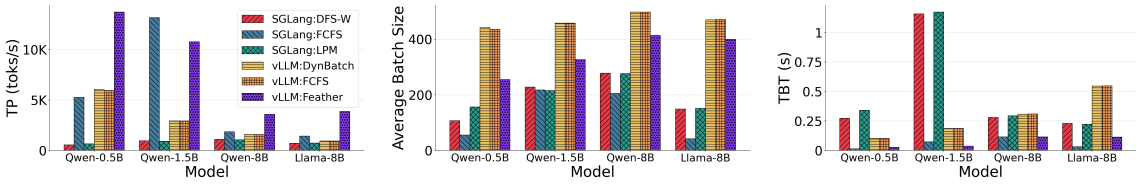
Figures 5.3(a) and 5.3(b) show throughput and average batch size as functions of decode tokens per request. Each decode requires a full sweep over the KV cache. The policies exhibit two regimes separated by a crossover around 20 decode tokens. In the prefill-dominated regime (1–20 tokens), all policies behave similarly: throughput increases modestly, and batch sizes converge as prefills limit the impact of locality-aware scheduling. Beyond 20 tokens, the policies diverge sharply. FEATHER scales strongly with decode length, reaching ∼3800 toks/s at 200 tokens—over 3× higher than SGLang policies. This gain arises from two factors. First, longer decode phases amplify locality across repeated KV cache sweeps. Second, a greater number of decode iterations allows the scheduler to accumulate more same-prefix requests, forming larger, more homogeneous batches. Together, these effects drive the steady increase in FEATHER's batch size and throughput. SGLang policies follow the opposite trajectory in batch size. Because shared prefix tokens count toward the token budget, each prefill consumes ∼5K tokens, while each decode step contributes only one token. As the decode length increases, fewer prefills occur per step, reducing the number of new requests for batching.

5.2.4 Varying Number of Shared Prefix Tokens

Figure 5.4 shows throughput across the length of the shared prefix. As expected, longer prefixes increase prefill compute and KV cache pressure, reducing throughput across policies. FEATHER's advantage grows with prefix length, driven by higher locality gains as sequences lengthen.

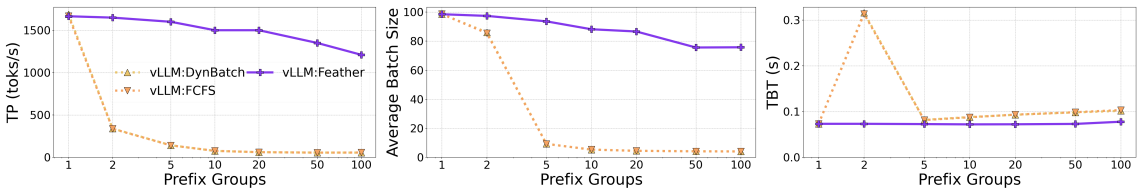
5.2.5 Varying Models

Figures 5.6(a), 5.6(b), 5.6(c) present throughput, average running batch size, and time between tokens (TBT) across models of varying sizes. The overall trend is consistent with expectations: smaller models achieve higher throughput owing to lower per-token compute costs. FEATHER outperforms all baselines across all models. The sole exception is Qwen-1.5B, where SGLang FCFS achieves marginally higher throughput; however, FEATHER compensates on the latency axis, exhibiting substantially lower TBT for the same model. A key structural observation is that FEATHER's relative throughput advantage over the baselines grows with model size. As model size increases, a larger fraction of the total execution time is spent on attention computation, which amplifies the benefits of temporal and spatial KV-cache locality that FEATHER exploits. This effect is visible in Figure 5.6(a): the gap between FEATHER and the next-best baseline widens from Qwen-0.5B to the 8B-scale models. The performance of SGLang's LPM and DFS-W schedulers degrades noticeably for smaller models. Both policies rely on CPU-side tree traversal, whose overhead scales with sequence length rather than model size; as the model shrinks, GPU execution time decreases while scheduler overhead remains roughly constant, causing the CPU



(a) Throughput (b) Average running batch size (c) Time between tokens (TBT)

Figure 5.6: Different Models



(a) Throughput (b) Average Batch Size (c) Time between tokens (TBT)

Figure 5.7: Varying number of prefix groups for LongChat 13B

to become an increasing bottleneck. FEATHER avoids this through its Chunked Hash Tree, which reduces per-step scheduler complexity and keeps CPU overhead negligible relative to GPU execution time across all model scales. Finally, Figure 5.6(c) highlights that vLLM's FCFS and DB baselines incur substantially higher TBT at larger model sizes, whereas Feather keeps TBT low and stable.

5.2.6 Results across LongChat Model

Figure 5.7 presents throughput, average batch size, and time between tokens (TBT) as the number of prefix groups varies for the LongChat 13B model. Overall, vLLM's FCFS and Dynamic Batching behave similarly across most configurations. For FCFS, there is a sharp drop in throughput when moving from 1 to 2 prefix groups, accompanied by an increase in TBT. This is largely due to the loss of locality benefits. The further decline from 2 to 5 prefix groups is driven by KV cache evictions, which is also reflected in the steep reduction in average batch size. Beyond this point, TBT stabilizes at a lower value, primarily because the batches have become much smaller. In contrast, FEATHER maintains stable and consistent throughput across all settings. At 100 prefix groups, it achieves up to 22× higher throughput than FCFS. By forming prefix-homogeneous batches, FEATHER not only improves scheduling efficiency but also significantly reduces KV cache evictions through more effective immediate cache reuse.

5.2.7 Varying Radix Tree Levels

Figure 5.5 shows throughput under different radix-tree sharing levels (seen in §3.2). All experiments use a fixed total sequence length of 4000 tokens per request. For baseline policies like vLLM's FCFS, going from 1 to 2 or more levels of the radix tree reduces the locality in KV cache traversal and, hence, degrades throughput. In contrast, FEATHER actively preserves prefix homogeneity within batches, leading to stable throughput at all levels.

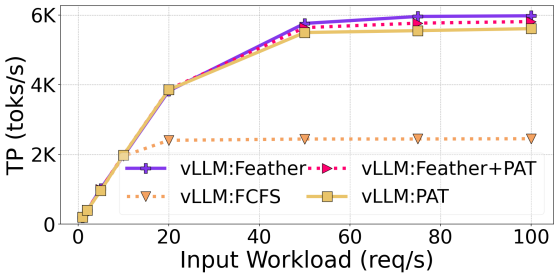


Figure 5.8: Feather vs PAT

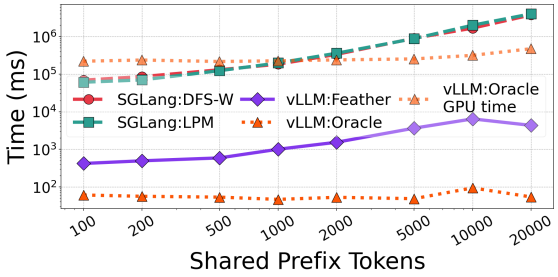


Figure 5.9: CPU overhead

5.2.8 Comparison with PAT

Figure 5.8 compares throughput across varying request rates for PAT, vLLM (FCFS), and Feather on an A100-80GB GPU. PAT is specifically optimized for this hardware through custom prefix-aware kernels. Despite this, Feather consistently matches or outperforms PAT across all request rates, demonstrating strong performance even on different hardware. This is particularly notable because Feather operates purely at the scheduling layer, without any kernel-level modifications. A naive integration of Feather and PAT performs quite similarly to Feather over most regimes. As future work, we aim to more tightly integrate Feather with prefix-aware attention kernels, enabling deeper co-design between the scheduler and kernel-level optimizations to further improve performance.

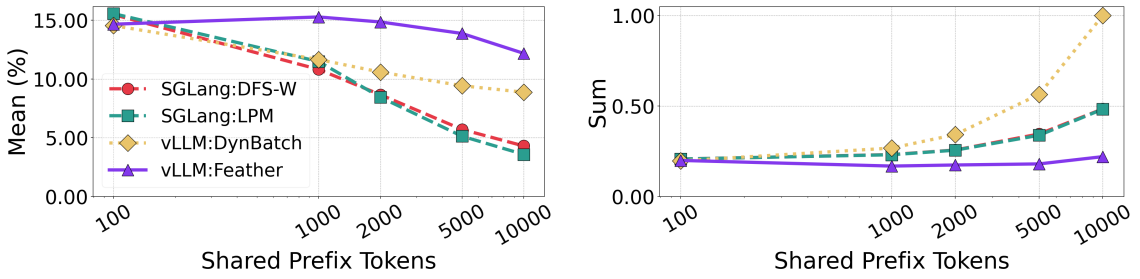
5.3 Micro-Benchmarks

5.3.1 Comparing Scheduler Compute Overhead across Different Policies

Figure 5.9 shows the CPU scheduling overhead for each policy across different shared prefix lengths. We include an Oracle policy where the caller supplies prefix group identifiers with each request. This allows the scheduler to batch same-prefix requests without any prefix matching overhead, resulting in effectively zero scheduling cost while preserving locality. We also show the GPU execution time for this Oracle scheduling policy. All three realistic policies—Feather, LPM, and DFS-W— incur higher overhead as the prefix length increases due to deeper radix-tree traversals and larger key comparisons. However, their growth rates diverge significantly. At 20K tokens, LPM and DFS-W incur roughly 1,000× higher overhead than Feather. At larger sequence lengths, their scheduling overhead exceeds the Oracle GPU execution time by up to 10×. This indicates that CPU overhead alone can negate the benefits of cache-aware batching. In contrast, Feather remains below 1% of GPU execution time across the entire range. We do a detailed analysis of the overheads of LPM and DFS-W policies below.

Complexity Analysis of LPM and DFS-W

We provide a detailed complexity analysis of the two SGLang cache-aware scheduling policies discussed above: Longest Prefix Match (LPM) and Depth-First Search with Weighting (DFS-W). Let W denote the number of requests in the waiting queue and T the maximum prefix length. For each request, LPM performs prefix matching on the global radix tree, costing $\mathcal{O}(T)$ per request, and then sorts the queue by matched prefix length, adding $\mathcal{O}(W \log W)$. The total per-round cost is therefore $\mathcal{O}(W \cdot T + W \log W)$. The result is overhead that scales *multiplicatively* with both queue size and context length. LPM implicitly acknowledges this by falling back to FCFS when $W > 128$ (which was disabled in our experiments). DFS-W avoids



(a) Mean (b) Sum

Figure 5.10: DRAM Bandwidth Utilization

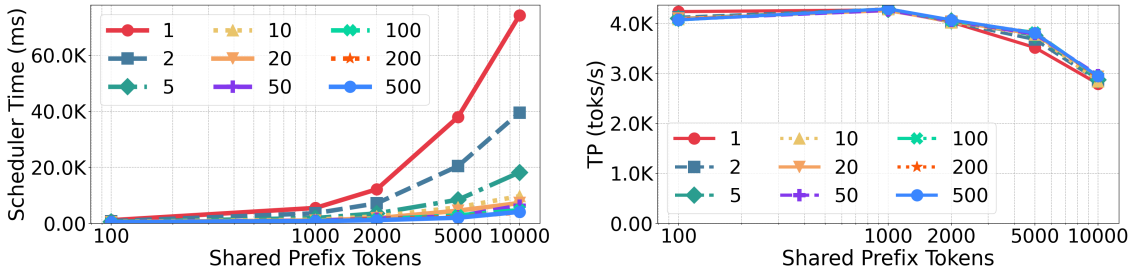
per-request traversal; instead, it performs a full traversal of the radix tree in each scheduling round. Let B be the total number of radix tree nodes. DFS-W first propagates subtree request counts upward in $\mathcal{O}(B)$, then re-traverses in a depth-first manner, sorting children by weight at every level. The total cost is $\mathcal{O}(W + B \log B)$. The key problem is that B reflects the server's accumulated history, not the current workload. The radix tree grows monotonically and is only pruned under memory pressure, so DFS-W's overhead increases over time, regardless of the current load. A server with a large accumulated cache ($B \gg W$) will incur high scheduling costs even during quiet periods. In summary, a burst of long-context requests penalizes LPM through the $W \cdot T$ term, while a large accumulated cache penalizes DFS-W through the $B \log B$ term. Either way, scheduling overhead can reach the same order of magnitude as GPU execution time. This demonstrates that identifying shared prefixes at runtime is fundamentally expensive and naive cache-aware scheduling can become the dominant bottleneck.

5.3.2 Mean DRAM Activity across Various Sequence Lengths

Figure 5.10(a) shows mean DRAM bandwidth utilization, while Figure 5.10(b) shows the total DRAM usage over the full inference duration (a proxy for total memory accessed) across policies and shared prefix lengths. FEATHER consistently achieves the highest bandwidth utilization due to improved spatial locality, leading to more effective prefetching. Furthermore, more homogeneous batches also increase temporal locality, which is reflected in lower total DRAM usage. SGLang's DFS-W and LPM exhibit lower mean DRAM utilization because of CPU scheduler stalls. However, they also achieve lower normalized total DRAM usage, as their prefix-aware scheduling occasionally forms homogeneous batches, allowing them to partially benefit from memory locality.

5.3.3 Effect of Chunk Size

Figures 5.11(a) and 5.11(b) show CPU overhead and throughput across shared prefix lengths for different chunk sizes in FEATHER's CHT. Increasing the chunk size reduces CPU overhead by enabling coarser-grained matching; a chunk size of 1 is equivalent to an exact radix tree. Overhead rises with prefix length due to more hash comparisons, while throughput declines due to higher KV cache memory pressure. At larger prefix lengths, the overhead gap between chunk size 1 and larger sizes widens substantially, yet throughput remains similar across all chunk sizes, suggesting that the CPU scheduling cost, even at the finest granularity, is not the primary bottleneck in our CHT. Overall, CHT is robust against chunk size selection, and even its radix tree-equivalent configuration (chunk size 1) achieves significantly better efficiency than a conventional radix tree.



(a) CPU Scheduler Overhead

(b) Throughput

Figure 5.11: Sensitivity to Chunk Size

Function	Number of Calls	Average Time (ns)	Total Time (ns)
Insert	5,000	153,714	768,570,573
AddToBatch	5,000	11,143	55,715,469
Finish	5,000	38,952	194,760,111
FindBest	26,405	6,641	175,372,986

Table 5.1: Time taken by individual functions of CHT

5.3.4 Time Taken by Individual Functions of CHT

Table 5.1 profiles CHT operations over an inference workload of $5K$ requests, each $5K$ tokens long. Insert, AddToBatch, and Finish are each invoked $5K$ times, once per request. FindBest, however, is called $26{,}405$ times because it is queried repeatedly during batch construction. The RL policy may reject candidates and trigger fresh lookups, and the scheduler may inspect the queue multiple times before committing to a Stop. Insert dominates total time because hash computation scans every token once, making it proportional to prompt length. Finish is more expensive than AddToBatch, despite their symmetric roles. This gap arises from Finish's additional forward tip-extension scan. After updating the working set, it traverses forward, level by level, to check if the shared prefix can be extended. In contrast, AddToBatch performs an early-terminating backward LCA scan, avoiding this overhead. FindBest is the cheapest per-call operation due to result caching and lazy heap traversal.

5.3.5 Convergence of Bandit Policy

We also evaluate how quickly Feather's bandit-based policy converges to a stable and efficient batching configuration. We construct a workload with five prefix groups, each containing 5K tokens, and issue requests at a rate of 5 per second. Figure 5.12 illustrates how the running batch size evolves over time for Feather under its bandit-based policy. At the beginning, the system enters an exploration phase, during which Feather experiments with different configurations and reaches relatively large batch sizes (around 500). However, these batches are heterogeneous. After this phase concludes (around 100 seconds), the system shifts to homogeneous batching, which yields higher throughput. As a result, Feather converges to a significantly smaller batch size and maintains it for the remainder of the experiment.

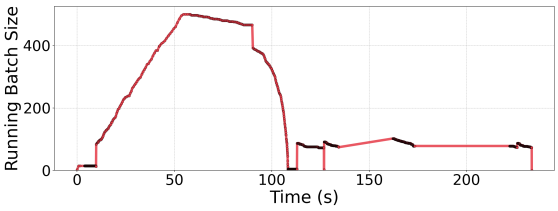


Figure 5.12: Convergence of Bandit Policy

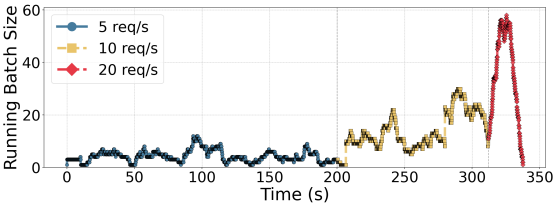
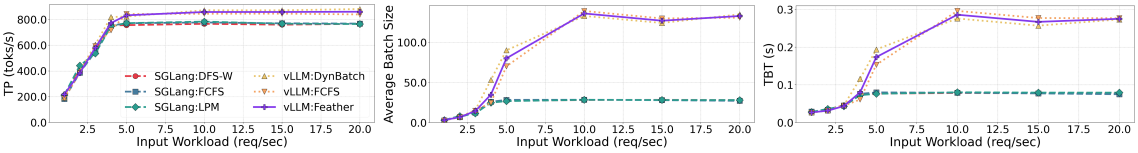


Figure 5.13: Performance of Bandit across Varying Workload



(a) Throughput (b) Average Batch Size (c) Time between tokens (TBT)

Figure 5.14: Workload with No Prefix Sharing

5.3.6 RL Policy across Varying Workload

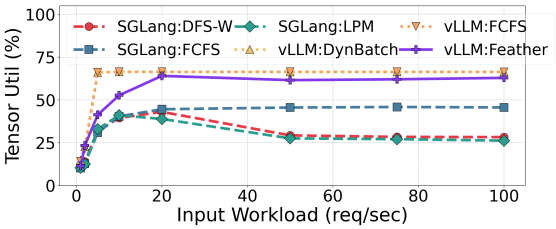
We generate a workload consisting of five prefix groups, each containing 5K tokens. The input request rate varies over time—from 5 to 10 to 20 requests per second—and we measure the average batch size as a function of time for FEATHER using its Bandit policy (Figure 5.13). From 0 to 200 seconds, the request rate is 5, and the batch size remains relatively low. Between 200 and 300 seconds, the rate increases to 10, leading to a noticeable rise in batch size. Finally, from 300 to 350 seconds, the request rate reaches 20, and the batch size peaks. (Fluctuations in the plot are due both to RL exploration and the workload being Poisson.) These results show that FEATHER dynamically adapts the batch size in response to both prefix homogeneity (i.e., the number of requests sharing the same prefix group) and the current workload intensity.

5.3.7 Workloads with No Prefix Sharing

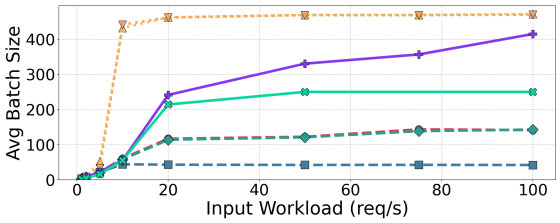
We next evaluate FEATHER and compare it against other scheduling policies on a workload without prefix sharing; that is, no two requests share a common prefix. Figure 5.14 reports throughput, average batch size, and time between tokens (TBT) across a range of input rates. At higher input rates, vLLM-based policies begin to outperform SGLang policies, primarily due to their ability to form larger batches. Notably, even in the absence of prefix sharing, FEATHER, powered by its adaptive RL-based policy—matches the performance of vLLM's FCFS scheduler. In contrast, SGLang policies achieve lower TBT, which can be attributed to their smaller batch sizes.

5.3.8 Tensor Core Utilization across Policies

Figure 5.15 compares tensor core utilization and average batch size across different request rates and scheduling policies. FEATHER shows slightly lower utilization at low request rates due to forming smaller batches, but it still delivers higher throughput at these points (as seen earlier in Figure 5.1(a)). As the workload increases, FEATHER gradually builds larger batches from prefix-homogeneous requests, and its utilization rises to match vLLM's FCFS. In contrast, SGLang's FCFS maintains relatively low utilization because it operates with consistently small batch sizes. Interestingly, even though SGLang's DFS-W and LPM achieve larger batch sizes, their utilization



(a) Tensor Core Utilization



(b) Average Batch Size

Figure 5.15: Input Workload

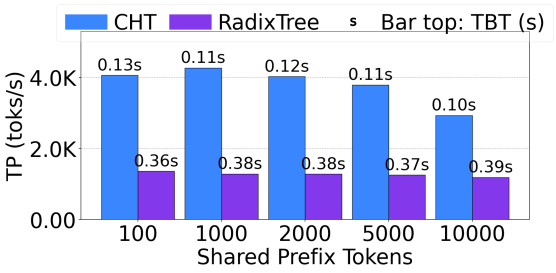


Figure 5.16: CHT vs. Radix Tree

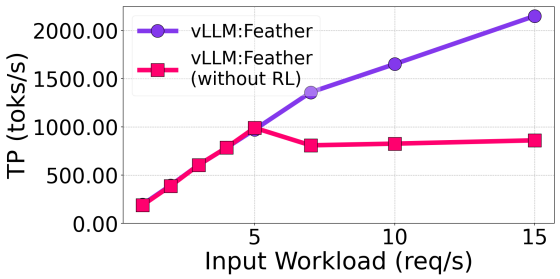


Figure 5.17: FEATHER: with and without RL

remains low. This is mainly due to CPU-side stalls between GPU executions, during which the GPU stays idle, effectively reducing overall utilization.

5.4 Ablation Study

5.4.1 What if we Replace the Chunked Hash Tree with a Radix Tree?

Figure 5.16 compares throughput and TBT when replacing CHT with a radix tree under the same RL policy. Across all prefix lengths, the radix tree yields much lower throughput. Despite homogeneous RL batching, higher CPU overhead causes frequent GPU stalls. This appears as higher TBT, indicating GPU idle time due to scheduling delays. CHT avoids this by reducing scheduling overhead, enabling higher GPU utilization and throughput.

5.4.2 What if We Remove RL?

Figure 5.17 compares throughput with and without RL at low input rates. Without RL, CHT still selects the best request, but batching never halts and always maximizes batch size. At <5 req/s, removing RL has little effect. Both fall back to FCFS, adding requests to keep GPUs utilized. Beyond this, the behaviors diverge. With RL, FEATHER detects homogeneous requests and stops batching at the right time, achieving higher throughput despite smaller average batch sizes. At higher rates than shown in the figure, large pools enable even non-RL to form ∼500-request homogeneous batches, leading the two curves to converge. These results show that FEATHER's RL policy makes load-aware decisions: it falls back to FCFS under low utilization to avoid GPU starvation, and closes batch formation when homogeneity is achievable, maximizing overall throughput.

6. Conclusion

Through our work, we showed that maximizing batch size alone is not sufficient for efficient LLM serving during the decode phase. Instead, we identified *prefix homogeneity*, the extent of the prefix shared across *all* requests in a batch, as a more important factor for improving memory locality and reducing redundant KV cache accesses. Higher batch sizes lead to better GPU utilization but also reduce the length of the prefix shared across all requests. Existing approaches ignore this aspect and also rely on expensive prefix detection mechanisms that introduce significant CPU overhead. To address these gaps, we proposed FEATHER, a lightweight prefix-aware scheduler that balances this trade-off between batch size and prefix homogeneity. It uses a Chunked Hash Tree (CHT) for fast prefix detection and a reinforcement learning-based batching policy that decides when to stop batch formation based on the loss in prefix homogeneity due to the addition of a new request. Our results show that FEATHER consistently improves end-to-end throughput across diverse workloads while keeping scheduling overhead low. As future work, we plan to extend this approach to distributed and multi-GPU settings. We are also looking into more adaptive and lightweight learning policies that can improve robustness across different workloads and deployment scenarios.

6.1 Source Code

The source code for this project is available at the following GitHub repository:

`https://github.com/sakshamrathi21/Structured_LLM_Optimization`

The repository contains all the necessary code files, scripts, and documentation to reproduce the experiments and results presented in this report.

A. Operations of Chunked Hash Tree

This section provides pseudo-codes and details of various algorithms used in the Chunked Hash Tree (§4.2).

A.1 Variables and Notations

Table A.1 provides a list of all the variables and notations used in the following pseudo-codes.

A.2 Prefix Hash Computation

The goal is to efficiently compute prefix-aligned hashes at fixed chunk boundaries. Algorithm 1 computes the hash vector for a token sequence using an incremental hashing scheme (Figure A.1). Instead of recomputing hashes for each prefix independently, the algorithm maintains a streaming hash state that is updated once per token. The algorithm processes each token exactly once and performs constant-time updates to the hash state. As a result, it runs in $\mathcal{O}(T)$ time and produces $C = \lceil T/K \rceil$ hash values, yielding $\mathcal{O}(C)$ output space. While Algorithm 1 processes tokens sequentially, the prefix-hash vector can equivalently be computed in parallel across chunks. The key property we want is that all requests sharing the first $c \cdot K$ tokens produce an identical h_c. This is achieved by partitioning the token sequence into chunks $S_c = (t_{(c-1)K+1}, \ldots, t_{cK})$, hashing each chunk independently in parallel, and then chaining the results via $h_c = \textsc{Hash}(h_{c-1} \parallel \textsc{Hash}(S_c))$, with h_0 set to a fixed initialization constant (Algorithm 2). With parallel workers, the dominant hashing phase runs in $\mathcal{O}(K)$ per worker rather than $\mathcal{O}(T)$ sequentially, reducing overall wall-clock time to $\mathcal{O}(K+C)$; the subsequent chaining pass costs only $\mathcal{O}(C)$ and is negligible in practice. This parallelization is most beneficial when K is large, as each chunk contains more tokens and therefore represents a heavier independent unit of work, allowing parallel workers to amortize scheduling overhead across a larger payload and yielding greater absolute speedup over the sequential baseline. Both algorithms produce a hash vector satisfying the prefix-consistency property—all requests sharing the first $c \cdot K$ tokens yield

Variable	Explanation
T	Sequence length in tokens
C	Number of chunks in a sequence
K	Chunk size
H	Hash vector
r	Request
W	Number of waiting requests
W_h	Average number of waiting requests sharing a hash
(l^*, h^*)	Shared prefix tip
h_l^r	Hash at level l of request r
`working_set`	Working set
`waiting`	Reverse index from hashes to waiting requests
`waiting_heap`	Min-heap
`request_hashes`	Map from request to its hash vector
`miss`	Map from request to its missing count
`active`	Set of requests present in the current batch
`waiting_requests`	Set of waiting requests
`ref`	Map from hash to number of active requests sharing it

Table A.1: Variables and Notations Used

the same h_c, yet the two vectors differ in their actual hash values, as Algorithm 2 is a distinct formulation rather than a parallelized execution of Algorithm 1.

A.3 Insertion

Algorithm 3 provides the pseudo-code for the INSERT procedure. Given a request with a token sequence, we compute its cumulative prefix-hash vector using COMPUTEHASHES and store it in a dictionary that maps each request to its hashes for easier reuse later (lines 1-2). For each level, we retrieve the corresponding hash and check whether the tuple is present in the working set; if not, the missing count is incremented (lines 6-7). Simultaneously, the request is inserted into the reverse index (line 9), which maps each hash to the set of waiting requests containing it and supports efficient lookup of prefix-sharing candidates. After processing all levels, we store the miss count in a map and insert the tuple of miss count and the request into the min-heap, which

Algorithm 1 COMPUTEHASHES(S, K)

Require: Token sequence $(t_1, \ldots, t_T)$; chunk size K
Ensure: Cumulative prefix-hash vector $H = (h_1, \ldots, h_C)$
1: $H \leftarrow []$; $C \leftarrow \lceil T/K \rceil$
2: Initialise incremental XXHASH-64 state s
3: **for** $i \leftarrow 1$ **to** T **do**
4: Feed t_i into s
5: **if** $i \bmod K = 0$ **or** $i = T$ **then**
6: H.APPEND(DIGEST(s))
7: **end if**
8: **end for**
9: **return** H

Algorithm 2 ComputeHashesParallel(S, K)

Require: Token sequence $(t_1, \ldots, t_T)$; chunk size K
Ensure: Cumulative prefix-hash vector $H = (h_1, \ldots, h_C)$

$C \leftarrow \lceil T/K \rceil$
$h_0 \leftarrow$ InitConst
parallel for $c \leftarrow 1$ **to** C **do** ▷ Independent per-chunk hashes
    $S_c \leftarrow (t_{(c-1)K+1}, \ldots, t_{\min(cK,T)})$
    $\ell_c \leftarrow \textsc{Hash}(S_c)$ ▷ Each chunk hashed independently
end parallel for
for $c \leftarrow 1$ **to** C **do** ▷ Sequential prefix chaining
    $h_c \leftarrow \textsc{Hash}(h_{c-1} \,\|\, \ell_c)$
end for
return $H = (h_1, \ldots, h_C)$

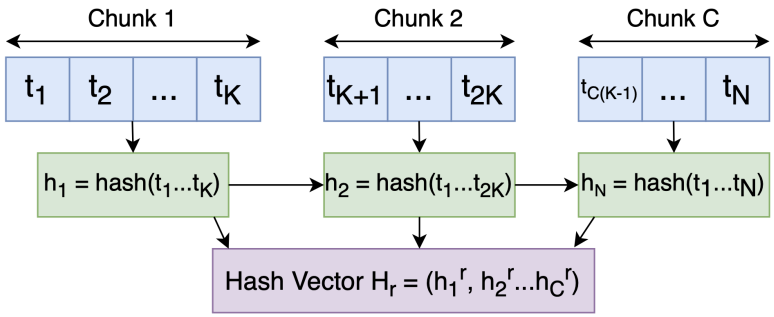


Figure A.1: Hash Computation in Chunked Hash Tree

is keyed by missing count, enabling efficient retrieval of the most compatible request (lines 11-12). The design avoids any updates to the working set or reference counts during insertion, ensuring low overhead.

A.4 Finding the Best Request

The FindBest procedure selects the most compatible waiting request by querying the min-heap while skipping stale entries arising from lazy updates until a valid minimum is found (lines 6-8). Once a candidate is identified, it is reinserted into the heap for future activations in case the RL policy decides to add this request to the batch (line 9), and three quantities are computed without modifying the system state: the current tip depth, the prospective depth, and the number of peers sharing the resulting prefix. The prospective depth is obtained via a backward scan over the cumulative hash vector, selecting the deepest level such that the corresponding level-hash tuple exists in the working set, which corresponds to the least common ancestor with the active set under the current state. The peer count is computed by counting waiting requests whose hash at that level matches that of the candidate request. The result is cached and reused until either the working set or the waiting queue changes, avoiding redundant recomputation across repeated calls. If no valid request is found, the procedure returns $\perp$.

Algorithm 3 INSERT(r, S)

Require: r; token sequence S; $working\_set$; $waiting$; $waiting\_heap$
Ensure: r registered and ready for scheduling
1: $H_r \leftarrow \textsc{ComputeHashes}(S, K)$
2: $request\_hashes[r] \leftarrow H_r$
3: $m_r \leftarrow 0$
4: **for** $\ell \leftarrow 1$ **to** $|H_r|$ **do**
5: $h \leftarrow h_\ell^r$
6: **if** $(\ell, h) \notin working\_set$ **then**
7: $m_r \leftarrow m_r + 1$
8: **end if**
9: $waiting[h] \leftarrow waiting[h] \cup \{r\}$
10: **end for**
11: $miss[r] \leftarrow m_r$
12: $waiting\_heap.\textsc{Push}(m_r,\ r)$

A.5 Adding a Request to the Active Batch

Given a request with its prefix-hash vector, the AddToBatch procedure promotes it to the active set (line 1) and updates the working set, reference counts, and other metadata in a single backward pass over the prefix-hash vector. For each level, if the reference count of the corresponding hash is zero (no other active request shares it), the level-hash tuple is inserted into the working set, and all corresponding waiting requests have their miss counts decremented and are reinserted into the waiting heap (lines 7-12). The reference counts are incremented at all levels (line 14). In parallel, the shared prefix tip is recomputed by tracking the deepest level at which the level-hash tuple remains in the working set; the first such match during the backward traversal determines the new tip (lines 15-17), while the loop continues to ensure that all state updates are applied. If this is the first active request, the tip is initialized directly to the end of its prefix-hash vector.

A.6 Removing a Request from the Active Batch

Given a completed request, the Finish procedure removes it from the active set (line 1) and updates the working set, reference counts, and other metadata. A forward pass over the prefix-hash vector decrements the reference count at each level (line 3); if it drops to zero, the level-hash tuple is removed from the working set, and all the corresponding waiting requests have their miss counts incremented and are reinserted into the min-heap (lines 4-10). After updating the working set, the shared prefix tip is recomputed by attempting to extend it forward: starting from the previous tip level, we check successive levels up to the minimum length among active requests and extend the tip whenever the reference count at that level equals the number of active requests, indicating agreement across all of them; the scan terminates at the first disagreement (lines 19-26). If the active set becomes empty, the tip is reset to $(0, \perp)$ (lines 12-14), and if only one request remains, the tip is set to its full length (lines 15-18). Finally, Remove is invoked to clean auxiliary structures such as the reverse index and the stored hash vector (line 27).

Algorithm 4 FindBest()

Require: $waiting\_heap$; $miss$; $active$; tip (ℓ^*, h^*); $working\_set$; $request\_hashes$
Ensure: $(r^*,\ \ell^*_{before},\ \ell^*_{after},\ peers)$ or $\perp$

1: **if** $waiting\_requests = \emptyset$ **then return** $\perp$
2: **end if**
3: **if** $cache\_valid$ **then return** $cached\_result$
4: **end if**
5: **while** $waiting\_heap \neq \emptyset$ **do**
6: $(m, r) \leftarrow waiting\_heap$.Pop()
7: **if** $r \in active$ **or** $r \notin miss$ **or** $miss[r] \neq m$ **then continue**
8: **end if**
9: $r^* \leftarrow r$; $waiting\_heap$.Push(m, r^*) ▷ restore for future calls
10: $\ell^*_{before} \leftarrow \ell^*$
11: $\ell^*_{after} \leftarrow \ell$ where ℓ is the largest $\ell \leq \min(\ell^*, |H_{r^*}|)$ s.t. $(\ell, h^{r^*}_{\ell}) \in working\_set$
12: $peers \leftarrow |\{w \in waiting\_requests : h^{w}_{\ell^*_{after}} = h^{r^*}_{\ell^*_{after}}\}|$
13: Cache and **return** $(r^*,\ \ell^*_{before},\ \ell^*_{after},\ peers)$
14: **end while**
15: **return** $\perp$

A.7 Batch Formation

BuildBatch describes the entire pipeline, where we incrementally construct a batch by repeatedly querying FindBest to obtain the most compatible candidate, along with summary statistics, which are combined with the current batch size to form the RL state (lines 3-6). A policy (heuristic, bandit, or Q-learning) maps this state to an action, either Add or Stop (line 7); if Add is selected, the request is activated via AddToBatch and appended to the batch; otherwise, the loop terminates (lines 8-10). The procedure continues until the waiting queue is exhausted or the policy decides to stop, after which the constructed batch is executed for one decode iteration to obtain the observed throughput (line 12). This throughput serves as the reward signal used to update the policy (line 13), with bandit methods updating their statistics and Q-learning applying a Bellman update with parameters α and γ, while heuristic policies incur no update.

A.8 Complexity Analysis

Table A.2 summarizes the time complexity of each operation. Tip maintenance in AddToBatch and Finish is subsumed within their $\mathcal{O}(C \cdot W_h \cdot \log W)$ passes and adds no asymptotic cost. In practice, the dominant cost is miss-count propagation, which updates all affected waiting requests per working-set change. Result caching avoids redundant FindBest traversals between mutations, keeping scheduling overhead low. We present the time taken by individual functions for a particular experimental setup in §5.3.4.

A.9 Find Best Request - Alternative Heuristic

An intuitive alternative to using the missing count (which is based on the difference with the working set) is to select a waiting request that maximizes the depth of the current shared prefix

Algorithm 5 AddToBatch(r)

Require: H_r; ref; *working_set*; *waiting*; *miss*; *waiting_heap*; tip (ℓ^*, h^*); *active*
Ensure: Updated ref, *working_set*, *miss*, (ℓ^*, h^*)

```
active ← active ∪ {r};   waiting_requests ← waiting_requests \ {r}
if |active| = 1 then
    (ℓ*, h*) ← (|H_r|, h^r_{|H_r|});   return
end if
ℓ_tip ← min(ℓ*, |H_r|);   (ℓ*, h*) ← (0, ⊥)
for ℓ ← |H_r| downto 1 do
    if ref[h^r_ℓ] = 0 then
        working_set ← working_set ∪ {(ℓ, h^r_ℓ)}
        for w ∈ waiting[h^r_ℓ], w ∉ active do
            miss[w] −= 1;
            waiting_heap.Push(miss[w], w)
        end for
    end if
    ref[h^r_ℓ] += 1
    if ℓ* = 0 and ℓ ≤ ℓ_tip and (ℓ, h^r_ℓ) ∈ working_set then
        (ℓ*, h*) ← (ℓ, h^r_ℓ) ▷ Least common ancestor found; continue loop for remaining working set updates
    end if
end for
```

Operation	Complexity
Insert	$\mathcal{O}(T + C \log W)$
AddToBatch	$\mathcal{O}(C \cdot W_h \cdot \log W)$
Finish	$\mathcal{O}(C \cdot W_h \cdot \log W)$
FindBest	$\mathcal{O}(\log W)$ amortized; $\mathcal{O}(1)$ cached
Remove	$\mathcal{O}(C)$

Table A.2: Time complexity of CHT scheduler operations

tip. However, this approach has a subtle flaw: the tip is inherently bottlenecked by the most divergent active request, limiting future extension. Consider a scenario where Level 1 is shared by all requests, but Level 2 splits into three branches: one for R_1, one shared by R_2 and R_3, and one for R_4 (Figure A.2). If the active batch contains R_1 and R_2, the shared prefix tip is restricted to Level 1. Selecting based only on this tip makes R_3 and R_4 appear equally viable. If the scheduler admits R_4, the tip remains at Level 1; when R_1 finishes, the remaining requests (R_2 and R_4) lie on divergent branches, leaving the tip still stuck. Our Chunked Hash Tree avoids this by evaluating candidates against the full working set using the missing count m_r, rather than anchoring to the bottlenecked tip. The working set tracks all prefix chunks present in any active request. In the example, since R_2 is active, its Level 2 chunk is already in the working set. Thus, R_3 requires fewer missing chunks than R_4, i.e., $m_{R_3} < m_{R_4}$. Selecting R_3 preserves future overlap: although the immediate tip remains bottlenecked, once R_1 finishes and is removed via Finish, the remaining batch $\{R_2, R_3\}$ shares a deeper prefix, and the tip extends to Level 2.

So, we have two ways to find the best waiting request to be added to the batch. One is based

Algorithm 6 FINISH(r)

Require: H_r; ref; $working\_set$; $waiting$; $miss$; $waiting\_heap$; tip (ℓ^*, h^*); $active$
Ensure: Updated ref, $working\_set$, $miss$, (ℓ^*, h^*)

$active \leftarrow active \setminus \{r\}$
for $\ell \leftarrow 1$ **to** $|H_r|$ **do**
  $\text{ref}[h_\ell^r] \mathrel{-}= 1$
  if $\text{ref}[h_\ell^r] = 0$ **then**
    $working\_set \leftarrow working\_set \setminus \{(\ell, h_\ell^r)\}$
    for $w \in waiting[h_\ell^r],\ w \notin active$ **do**
      $miss[w] \mathrel{+}= 1$;
      $waiting\_heap$.PUSH$(miss[w], w)$
    end for
  end if
end for
if $active = \emptyset$ **then**
  $(\ell^*, h^*) \leftarrow (0, \perp)$; **return**
end if
if $|active| = 1$ **then**
  let q be the sole element of $active$
  $(\ell^*, h^*) \leftarrow (|H_q|,\ h^q_{|H_q|})$; **return**
end if
while $\ell^* + 1 \le \min_{q \in active} |H_q|$ **do** ▷ try to extend tip forward
  $\ell \leftarrow \ell^* + 1$
  if $\text{ref}[h_\ell^{q_0}] = |active|$ for arbitrary $q_0 \in active$ **then** ▷ all active requests agree at level ℓ
    $(\ell^*, h^*) \leftarrow (\ell,\ h_\ell^{q_0})$
  else
    break
  end if
end while
REMOVE(r) ▷ cleans reverse index and request hashes

on the difference from the working set, and the other is based on the shared prefix tip. Next, we try to formalize the metrics on which both of these ways are based. Let W be the hashes in the current working set, which can be defined as: $\cup_{r \in A} H^r$, where A is the active set of requests and H^r is the set of hashes for request r. CHT selects the request that has the minimum miss-count, which can be defined as $m_W(r) = |H^r \setminus W|$. The shared prefix tip is the deepest level shared by all the requests of the active batch, which can be defined as $\max\{\ell : \forall r \in A, (\ell, h_\ell^r) \in W\}$. The alternative heuristic selects the request that leads to the maximum depth of the shared prefix tip, which is equivalent to minimizing the number of hashes missing from the working set at or above the current tip for that request. Let $m_S(r) = |S \setminus H^r|$, where S is the set of all hashes at or above the current tip level in the working set, i.e., $S = \{h : (\ell, h) \in W, \ell \le \ell^*\}$, where ℓ^* is the current tip level. That is, $m_S(r)$ denotes the number of hash chunks present in S, but not present in the request r. Therefore, we explore the alternative heuristic that minimizes $m_S(r)$ instead of $m_W(r)$.

Algorithm 7 BUILDBATCH()

Require: Waiting queue Q; policy π (heuristic, bandit, or Q-learning); discount γ; learning rate α
Ensure: Batch B ready for one decode iteration

```
B ← ∅
while Q ≠ ∅ do
    (r*, c_before, c_after, w) ← FINDBEST()
    if r* = ⊥ then break
    end if
    s ← (|B|, Δ = c_before − c_after, w)
    a ← π(s)                          ▷ heuristic rule / UCB look-up / ε-greedy Q-table
    if a = STOP then break
    end if
    ADDTOBATCH(r*);   B ← B ∪ {r*}
end while
throughput_decode ← EXECUTEBATCH(B)
Update π with observed reward throughput_decode          ▷ no-op for heuristic
return B
```

Let us assume that all the requests have a length of C chunks, and the best requests emitted from the two metrics are r_W and r_S, respectively. Let us also assume that the shared prefix tip is at level ℓ^*. We aim to show that $m_S(r_W) = m_S(r_S)$, that is, the working set metric causes equal loss in the shared prefix tip, implying that r_W does not lead to less depth in the shared prefix tip compared to r_S. Let us assume, to the contrary, that $m_S(r_S) < m_S(r_W)$. Now, we compare $m_W(r_W)$ and $m_W(r_S)$ and aim to show that $m_W(r_W) > m_W(r_S)$, which contradicts the fact that r_w minimizes the metric m_W.

Since S represents the shared prefix of all active requests up to depth ℓ^*, the working set W contains exactly one unique hash per level for all levels $\ell \leq \ell^*$, which constitutes the set S. This property dictates that if any request x misses a hash in S at some level $k < \ell^*$, its prefix diverges from the *entire* active batch at level k. Consequently, due to the cumulative nature of the chunked hashes, x cannot share any hashes with W at any level $\geq k$. Specifically, if a request does not fully match the shared tip (i.e., $m_S(x) > 0$), it cannot overlap with any deeper divergent branches in W. For such a request, its overlap with the working set is exactly its overlap with the shared tip: $|H^x \cap W| = |H^x \cap S| = \ell^* - m_S(x)$. Using our contrary assumption $m_S(r_S) < m_S(r_W)$, we evaluate the cost of both requests under the working set metric, $m_W(x) = C - |H^x \cap W|$. We consider two cases for r_S:

Case 1: r_S fully matches the shared prefix tip ($m_S(r_S) = 0$). Since r_S matches S entirely, $|H^{r_S} \cap S| = \ell^*$. It may also overlap with deeper branches in W, meaning $|H^{r_S} \cap W| \geq \ell^*$. Therefore, its cost is $m_W(r_S) \leq C - \ell^*$. Because $m_S(r_W) > m_S(r_S) = 0$, r_W diverges before the tip. As established, this means $|H^r_W \cap W| = \ell^* - m_S(r_W) < \ell^*$. Thus, $m_W(r_W) = C - |H^r_W \cap W| > C - \ell^*$. Combining these inequalities yields: $m_W(r_S) < m_W(r_W)$ (Figure A.3(a)).

Case 2: r_S diverges before the shared prefix tip ($m_S(r_S) > 0$). Since $m_S(r_W) > m_S(r_S) > 0$, both requests diverge before ℓ^*. Their overlap with the working set is strictly limited to their overlap with S. Thus, $|H^{r_S} \cap W| = \ell^* - m_S(r_S)$ and $|H^r_W \cap W| = \ell^* - m_S(r_W)$. Because

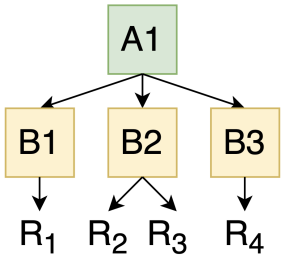
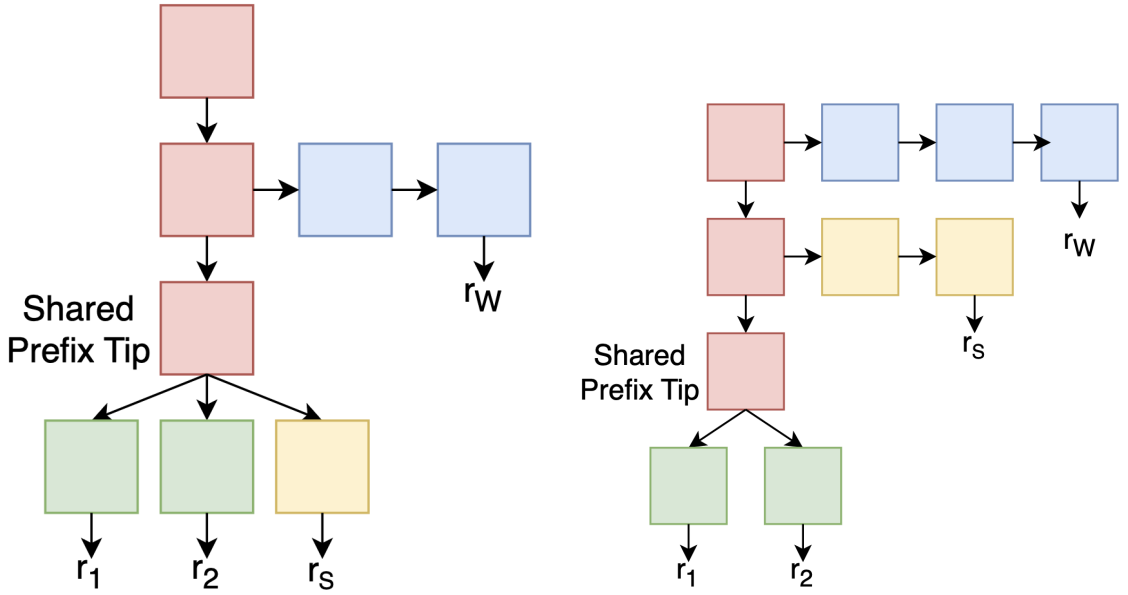


Figure A.2: FindBest - Alternative Heuristic Example

$m_S(r_S) < m_S(r_W)$, it logically follows that $|H^{r_S} \cap W| > |H^r_W \cap W|$. Consequently: $m_W(r_S) < m_W(r_W)$ (Figure A.3(b)).

In both cases, assuming $m_S(r_S) < m_S(r_W)$ unavoidably leads to $m_W(r_S) < m_W(r_W)$. However, this directly contradicts our initial premise that r_W is the optimal request chosen by the Chunked Hash Tree metric (which guaranties $m_W(r_W) \leq m_W(q)$ for all waiting requests q). Therefore, our assumption must be false, proving that $m_S(r_W) \leq m_S(r_S)$. Because r_S is defined as the absolute minimum for m_S, it must strictly hold that $m_S(r_W) = m_S(r_S)$. This completes the proof that optimizing for the working set metric is mathematically guaranteed to be at least as optimal as the shared prefix tip metric for preserving the immediate tip depth while simultaneously exposing deeper amortization opportunities in $W \setminus S$.



(a) Case 1 (b) Case 2

Figure A.3: FindBest cases: r_1 and r_2 are both present in the active batch, and hence the working set is the union of all of their chunks. The shared prefix tip is at the third level, and the set S of all shared prefix chunks are marked in red. In the first case, r_S fully matches the shared prefix tip, whereas r_W hangs from the middle. For this case, $m_W(r_W) = 2, m_W(r_S) = 1, m_S(r_W) = 1, m_S(r_S) = 0$. In the second case, both the requests r_S and r_W do not fully match the shared prefix tip, however r_W hangs off from an earlier chunk. For this case, $m_W(r_W) = 3, m_W(r_S) = 2, m_S(r_W) = 2, m_S(r_S) = 1$. Therefore, for both the cases, m_W and m_S exhibit similar behaviour.

Bibliography

[1] Jason Wei et al. "Emergent Abilities of Large Language Models". In: Transactions on Machine Learning Research (2022). Survey Certification. ISSN: 2835-8856. URL: https://openreview.net/forum?id=yzkSU5zdwD (cited on page 9).

[2] OpenAI et al. GPT-4 Technical Report. 2024. arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774 (cited on page 9).

[3] DeepSeek-AI et al. DeepSeek-V3 Technical Report. 2025. arXiv: 2412.19437 [cs.CL]. URL: https://arxiv.org/abs/2412.19437 (cited on page 9).

[4] Sumit Kumar Dam et al. A Complete Survey on LLM-based AI Chatbots. 2024. arXiv: 2406.16937 [cs.CL]. URL: https://arxiv.org/abs/2406.16937 (cited on page 9).

[5] Xin Luna Dong et al. "Towards Next-Generation Intelligent Assistants Leveraging LLM Techniques". In: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, 2023. DOI: 10.1145/3580305.3599572 (cited on page 9).

[6] Binyuan Hui et al. Qwen2.5-Coder Technical Report. 2024. arXiv: 2409.12186 [cs.CL]. URL: https://arxiv.org/abs/2409.12186 (cited on page 9).

[7] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: Advances in Neural Information Processing Systems. Curran Associates, Inc., 2020. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf (cited on page 9).

[8] Jiayi Yao et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion". In: Proceedings of the Twentieth European Conference on Computer Systems. Association for Computing Machinery, 2025. DOI: 10.1145/3689031.3696098 (cited on page 9).

[9] OpenAI. ChatGPT. https://chatgpt.com. 2024 (cited on page 9).

[10] Anthropic. Claude. https://claude.ai. 2024 (cited on page 9).

[11] Google. Gemini. https://gemini.google.com. 2024 (cited on page 9).

[12] Bowen Pang, Kai Li, and Feifan Wang. "Optimizing LLM Inference Throughput via Memory-aware and SLA-constrained Dynamic Batching". In: arXiv preprint arXiv:2503.05248 (2025) (cited on pages 9, 42).

[13] Yinmin Zhong et al. "DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving". In: Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation. USENIX Association, 2024 (cited on pages 9, 18).

[14] Wanyi Zheng et al. BucketServe: Bucket-Based Dynamic Batching for Smart and Efficient LLM Inference Serving. 2025. arXiv: 2507.17120 [cs.DC]. URL: https://arxiv.org/abs/2507.17120 (cited on pages 9, 18).

[15] Amey Agrawal et al. "Taming throughput-latency tradeoff in LLM inference with sarathi-serve". In: Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation. USENIX Association, 2024 (cited on pages 9, 18).

[16] Qidong Su et al. Seesaw: High-throughput LLM Inference via Model Re-sharding. 2025. arXiv: 2503.06433 [cs.DC]. URL: https://arxiv.org/abs/2503.06433 (cited on page 9).

[17] Bingyang Wu et al. "LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism". In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. Association for Computing Machinery, 2024. DOI: 10.1145/3694715.3695948 (cited on page 9).

[18] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: Proceedings of the 29th Symposium on Operating Systems Principles. Association for Computing Machinery, 2023. DOI: 10.1145/3600006.3613165 (cited on pages 9–11, 14, 18–20, 27, 40, 42).

[19] Ramya Prabhu et al. "vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention". In: Association for Computing Machinery, 2025. DOI: 10.1145/3669940.3707256 (cited on page 9).

[20] Chen Zhang et al. "Jenga: Effective Memory Management for Serving LLM with Heterogeneity". In: Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles. Association for Computing Machinery, 2025. DOI: 10.1145/3731569.3764823 (cited on page 9).

[21] Jerry Chee et al. "QuIP: 2-bit quantization of large language models with guarantees". In: Proceedings of the 37th International Conference on Neural Information Processing Systems. Curran Associates Inc., 2023 (cited on page 9).

[22] Ji Lin et al. "AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration". In: (2025). DOI: 10.1145/3714983.3714987 (cited on page 9).

[23] Jinjun Yi et al. "PAT: Accelerating LLM Decoding via Prefix-Aware Attention with Resource Efficient Multi-Tile Kernel". In: Association for Computing Machinery, 2026. DOI: 10.1145/3779212.3790200 (cited on pages 9, 19, 34, 41, 42).

[24] Zaifeng Pan et al. "FastTree: Optimizing Attention Kernel and Runtime for Tree-Structured LLM Inference". In: Proceedings of Machine Learning and Systems. MLSys, 2025. URL: https://proceedings.mlsys.org/paper_files/paper/2025/file/96894468eb44631a32d7ebd56f9892c7-Paper-Conference.pdf (cited on pages 9, 19, 25, 26, 42).

[25] Aditya K. Kamath et al. "POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference". In: Association for Computing Machinery, 2025. DOI: 10.1145/3676641.3715996 (cited on pages 9, 18).

[26] Zejia Lin et al. "Bullet: Boosting GPU Utilization for LLM Serving via Dynamic Spatial-Temporal Orchestration". In: Association for Computing Machinery, 2026. DOI: 10.1145/3779212.3790135 (cited on pages 9, 18).

[27] Tom B. Brown et al. "Language models are few-shot learners". In: Proceedings of the 34th International Conference on Neural Information Processing Systems. Curran Associates Inc., 2020 (cited on pages 9, 18).

[28] Ethan Perez, Douwe Kiela, and Kyunghyun Cho. "True Few-Shot Learning with Language Models". In: Advances in Neural Information Processing Systems. Curran Associates, Inc., 2021. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/5c04925674920eb58467fb52ce4ef728-Paper.pdf (cited on pages 9, 18).

[29] The Big Prompt Library Contributors. The Big Prompt Library: A Collection of Prompts, System Prompts and LLM Instructions. https://github.com/0xeb/TheBigPromptLibrary. GitHub repository. 2024 (cited on pages 9, 18).

[30] Patrick Lewis et al. "Retrieval-augmented generation for knowledge-intensive NLP tasks". In: Proceedings of the 34th International Conference on Neural Information Processing Systems. Curran Associates Inc., 2020 (cited on pages 9, 18).

[31] Lianmin Zheng et al. "SGLang: efficient execution of structured language model programs". In: Proceedings of the 38th International Conference on Neural Information Processing Systems. Curran Associates Inc., 2024 (cited on pages 9–11, 18, 23–25, 29, 33, 40, 42).

[32] Jinwei Yao et al. "DeFT: Decoding with Flash Tree-attention for Efficient Tree-structured LLM Inference". In: International Conference on Learning Representations. 2024. URL: https://api.semanticscholar.org/CorpusID:268819748 (cited on pages 9, 19).

[33] Lu Ye et al. "ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition". In: Association for Computational Linguistics, 2024. DOI: 10.18653/v1/2024.acl-long.623 (cited on pages 9, 19).

[34] "Understanding the Workload Characteristics of Large Language Model Development". In: (). URL: https://www.usenix.org/publications/loginonline/understanding-workload-characteristics-large-language-model-development (cited on page 12).

[35] "How LLMs Generate Text for the Rest of Us". In: (). URL: https://pm.dartus.fr/posts/2025/how-llm-generate-text/ (cited on pages 14, 17).

[36] IBM. "SysML Workshop". In: 2025 (cited on page 16).

[37] Ivy Bo Peng et al. "Exploring the Performance Benefit of Hybrid Memory System on HPC Environments". In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2017. DOI: 10.1109/ipdpsw.2017.115 (cited on page 18).

[38] vLLM Project Contributors. Automatic Prefix Caching. https://docs.vllm.ai/en/stable/design/prefix_caching/. vLLM Documentation. Accessed: 2026-03-10. 2024 (cited on page 18).

[39] Zhen Zheng et al. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-orien 2025. arXiv: 2412.03594 [cs.CL]. URL: https://arxiv.org/abs/2412.03594 (cited on pages 18, 29, 33, 34).

[40] Yilong Zhao et al. "BlendServe: Optimizing Offline Inference with Resource-Aware Batching". In: Association for Computing Machinery, 2026. DOI: 10.1145/3779212.3790133 (cited on pages 18, 34).

[41] "vLLM Prefix Caching". In: (). URL: https://docs.vllm.ai/en/stable/design/prefix_caching.html (cited on page 22).

[42] AI@Meta. "Llama 3 Model Card". In: (2024). URL: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md (cited on pages 27, 41).

[43] NVIDIA Corporation. NVIDIA RTX 6000 Ada Generation Datasheet. Accessed: January 25, 2026. 2024. URL: https://resources.nvidia.com/en-us-briefcase-for-datasheets/proviz-print-rtx6000-1?ncid=no-ncid (visited on 01/25/2026) (cited on pages 27, 41).

[44] Chenxin An et al. "L-Eval: Instituting Standardized Evaluation for Long Context Language Models". In: Association for Computational Linguistics, 2024. DOI: 10.18653/v1/2024.acl-long.776 (cited on pages 27, 41).

[45] Tri Dao et al. "FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness". In: Proceedings of the 36th International Conference on Neural Information Processing Systems. Curran Associates Inc., 2022 (cited on page 27).

[46] NVIDIA Corporation. NVIDIA Data Center GPU Manager (DCGM) Documentation. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/. 2026 (cited on page 28).

[47] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multi-armed Bandit Problem". In: Machine Learning (2002). DOI: 10.1023/A:1013689704352 (cited on page 39).

[48] Christopher J. C. H. Watkins and Peter Dayan. "Q-Learning". In: Machine Learning (1992). DOI: 10.1007/BF00992698 (cited on page 40).

[49] Jinze Bai et al. "Qwen Technical Report". In: arXiv preprint arXiv:2309.16609 (2023) (cited on page 41).

[50] Dacheng Li et al. How Long Can Open-Source LLMs Truly Promise on Context Length? June 2023. URL: https://lmsys.org/blog/2023-06-29-longchat (cited on page 41).

[51] Yushi Bai et al. "LongBench v2: Towards Deeper Understanding and Reasoning on Realistic Long-context Multitasks". In: Association for Computational Linguistics, 2025. DOI: 10.18653/v1/2025.acl-long.183 (cited on page 41).