Accelerating LLM Inference via ANN-Based Draft Model Construction

Speculative decoding with dense and FlashHead draft models

Master's thesis in Computer science and engineering

Guangyu Ma and Weiyou Wang

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Accelerating LLM Inference via ANN-Based Draft Model Construction

Speculative decoding with dense and FlashHead draft models

Guangyu Ma and Weiyou Wang

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Accelerating LLM Inference via ANN-Based Draft Model Construction
Speculative decoding with dense and FlashHead draft models
Guangyu Ma and Weiyou Wang

© Guangyu Ma and Weiyou Wang, 2026.

Supervisor: Matti Karppa, Department of Computer Science and Engineering
Advisor: TODO
Examiner: TODO

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2026

Accelerating LLM Inference via ANN-Based Draft Model Construction
Speculative decoding with dense and FlashHead draft models
Guangyu Ma and Weiyou Wang
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Large language model inference is limited by the sequential nature of autoregressive decoding. Speculative decoding reduces this cost by using a smaller draft model to propose several candidate tokens, which are then verified by a larger target model. However, this shifts part of the work to the draft side: every proposed token still requires a full-vocabulary output projection, which can become a non-negligible source of latency.

This thesis studies whether an approximate-nearest-neighbor based replacement for this output projection can reduce the draft-side cost in speculative decoding. We use an existing FlashHead model as a drop-in replacement for the dense projection of a Qwen3-0.6B draft model, while keeping the target model and the speculative verification procedure unchanged. The final experimental setup focuses on a Qwen3-32B-AWQ base model and compares three decoding modes: baseline decoding, speculative decoding with a dense draft model, and speculative decoding with a FlashHead draft model.

The results show that FlashHead strongly accelerates the isolated draft projection, reducing single-step projection latency from 2.493 ms to 0.577 ms, a 4.321× speedup. At the complete draft-model step level, this becomes a smaller 1.108× speedup, and in the ordinary end-to-end speculative comparison FlashHead improves throughput only from 9.769 to 9.823 tokens/s relative to the dense draft. With deferred final-token verification, FlashHead reaches 11.374 tokens/s compared with 10.731 tokens/s for the dense draft, showing that draft-side projection acceleration is real but mostly diluted by target-side verification and finalization costs.

Keywords: Large Language Models, Speculative Decoding, Approximate Nearest Neighbor, FlashHead

Acknowledgements

TODO: Add acknowledgements before submission.

Guangyu Ma and Weiyou Wang, Gothenburg, 2026-05-17

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem . . . 1
- 1.3 Aim and Research Questions . . . 2
- 1.4 Scope . . . 2
- 1.5 Overview of Results . . . 2
- 1.6 Thesis Structure . . . 2

2 Theory **5**
- 2.1 Autoregressive LLM Inference . . . 5
- 2.2 Speculative Decoding . . . 6
- 2.3 Dense Unembedding Layer . . . 8
- 2.4 Approximate Nearest Neighbor Search . . . 9
- 2.5 FlashHead . . . 9

3 Methods **13**
- 3.1 Overview . . . 13
- 3.2 Models . . . 13
- 3.3 Speculative Decoding Implementation . . . 13
- 3.4 FlashHead Integration . . . 14
- 3.5 Diagnostic Profiling . . . 15
- 3.6 Optimization of the Speculative Loop . . . 16
  - 3.6.1 Pipelined finalization and proposal . . . 16
  - 3.6.2 Deferred final-token verification . . . 16
- 3.7 Benchmark Protocol . . . 18

4 Experimental Results **21**
- 4.1 Benchmark Dataset . . . 21
- 4.2 Evaluation Metrics . . . 21
- 4.3 Model-only Results for the 0.6B Draft Model . . . 22
  - 4.3.1 Dense LM Head vs. FlashHead . . . 22
  - 4.3.2 0.6B Dense Model vs. 0.6B Flash Model . . . 22
- 4.4 Speculative Decoding with FlashHead . . . 23
- 4.5 Profiling Results and Implications . . . 24
- 4.6 Deferred Final-token Verification for Dense and FlashHead . . . 25

5 Conclusion **27**
- 5.1 Summary . . . 27

5.2 Conclusions 27
5.3 Limitations and Future Work 28

Bibliography **31**

A Experiment Artifacts **I**

List of Figures

2.1 Autoregressive decoding flow for the Qwen3-32B target model. 6
2.2 Speculative decoding with a draft model that proposes candidate tokens and a target model that verifies the proposed continuation. . . 7
2.3 Dense unembedding scoring compared with FlashHead's cluster-first retrieval and reduced candidate scoring. 10

3.1 Deferred final-token verification. The final target token emitted in one speculative round is carried into the next target verification batch, allowing the target cache to advance without a separate single-token target forward for most rounds. 17

List of Tables

4.1 Single-step head latency of the dense LM head and FlashHead. 22
4.2 Head-level speedup from the dense LM head to FlashHead. 22
4.3 Complete single-step decode latency of the 0.6B dense and Flash models. 22
4.4 Complete model-level speedup from the dense model to the Flash model. 23
4.5 End-to-end speculative decoding results with dense and FlashHead draft models. 23
4.6 Decode-throughput speedups in speculative decoding. 23
4.7 Phase-level profile of dense, dense-shared, and FlashHead draft modes. 24
4.8 Profiling diagnosis and supporting evidence. 24
4.9 End-to-end comparison with deferred verification enabled for dense and FlashHead draft models. 25
4.10 Decode-throughput speedups with deferred verification. 25
4.11 Draft-path comparison under deferred verification. 25

1

Introduction

1.1 Motivation

Large language models (LLMs) have become a central technology for text generation, interactive assistants, code generation, and information processing. As model capability improves, model size and deployment demand also increase. This makes inference efficiency a practical bottleneck: a deployed system must repeatedly generate tokens under latency, memory, and throughput constraints [1].

Autoregressive decoding is especially costly because tokens are generated sequentially. Even when key-value (KV) caching avoids recomputing past attention states, each new token still requires a forward pass and an output projection over the vocabulary. System-level optimizations such as paged attention and FlashAttention reduce parts of the memory and attention cost, but they do not remove the sequential dependency of generation or the repeated language-modeling head computation [2], [3], [4].

Speculative decoding is one approach to reducing the cost of autoregressive inference [5], [6]. Instead of asking the large target model to generate every token one by one, a smaller draft model proposes a block of candidate tokens. The target model verifies these candidates in a batched step and accepts the longest valid prefix. If the draft model is accurate enough and cheap enough, several output tokens can be produced for the cost of fewer target-model invocations.

1.2 Problem

The speedup of speculative decoding depends not only on the target model, but also on the cost and quality of the draft model. The draft model is invoked repeatedly during generation, and each draft step normally ends with a dense language-modeling head that scores the entire vocabulary. For a hidden state $h_t \in \mathbb{R}^d$ and a vocabulary embedding matrix $E \in \mathbb{R}^{v \times d}$, where v is the vocabulary size and d is the hidden size, this projection requires work proportional to vd.

This thesis investigates whether this draft-side cost can be reduced by replacing the dense head with an approximate-nearest-neighbor (ANN) based head. In particular, we use an existing FlashHead model [7] as a practical ANN-style replacement for the dense language-modeling head. The target model, draft-model transformer body, and speculative verification procedure are kept unchanged so that the comparison focuses on the effect of replacing the draft head.

1.3 Aim and Research Questions

The aim of this thesis is to evaluate whether an ANN-based draft head can improve LLM inference efficiency in a speculative decoding pipeline.

The thesis is guided by the following research questions:

1. How does a FlashHead draft model affect the cost of draft token proposal compared with a dense draft model?
2. In a speculative decoding setup with a Qwen3-32B-AWQ base model, how do baseline decoding, dense-draft speculative decoding, and FlashHead-draft speculative decoding compare?
3. How do throughput, latency, and acceptance behavior interact when the dense draft head is replaced by FlashHead?
4. Under the implemented decoding protocol, does a local reduction in draft-head cost translate into an end-to-end inference improvement?

1.4 Scope

The final experimental scope is limited to a controlled comparison using Qwen3 models. The base model is Qwen3-32B-AWQ, the dense draft model is Qwen3-0.6B, and the FlashHead draft model is an existing Qwen3-0.6B-FlashHead model. The implementation follows the decoding behavior used in the experimental codebase.

This work does not train a new FlashHead model, does not modify the base model, and does not change the speculative verification logic.

1.5 Overview of Results

The final results show a clear local but limited system-level effect. FlashHead reduces isolated head latency from 2.493 ms to 0.577 ms, a 4.321× speedup, and reduces the complete 0.6B draft-model step from 18.330 ms to 16.550 ms, a 1.108× speedup. In the ordinary speculative loop, however, dense and FlashHead drafts reach 9.769 and 9.823 tokens/s respectively, only a 1.006× end-to-end improvement. With deferred final-token verification, FlashHead reaches 11.374 tokens/s compared with 10.731 tokens/s for the dense draft, a 1.060× gain. The main interpretation is that target-side verification, finalization, and the draft transformer body dominate the pipeline, so reducing draft-head cost helps but is not sufficient for a large end-to-end speedup.

1.6 Thesis Structure

Chapter 2 introduces the relevant background on autoregressive decoding, speculative decoding, dense LM heads, and ANN-style head replacement. Chapter 3 describes the implemented method, models, benchmark framework, and experimental protocol.

Chapter 4 reports the final measurements. Chapter 5 summarizes the conclusions, limitations, and future work.

2

Theory

2.1 Autoregressive LLM Inference

Decoder-only large language models generate text autoregressively. Given a prompt and a partial output sequence $x_{<t}$, the model estimates the conditional distribution $p_\theta(x_t \mid x_{<t})$, selects a next token x_t, appends it to the sequence, and repeats the same procedure until an end-of-sequence token or a length limit is reached. This token-by-token dependency is the central constraint in autoregressive inference: the model cannot compute the hidden state for token $t + 1$ before token t has been selected.

Inference is commonly divided into two phases. In the prefill phase, the model processes all prompt tokens in parallel and builds the key-value (KV) cache used by the attention layers. In the decode phase, each new token is generated by running the transformer on the most recent token while reusing the cached keys and values for previous tokens. The final hidden state is then projected to vocabulary logits by the unembedding layer. A decoding rule then selects the next token; greedy decoding chooses the highest-scoring token, while sampling draws from a probability distribution. The KV cache is updated for the next step. KV caching therefore removes redundant recomputation over the prefix, but it does not remove the need for one sequential model step per generated token. Figure 2.1 illustrates this autoregressive decoding loop for the Qwen3-32B target model.

The unembedding layer, often called the language-modeling (LM) head in implementation-oriented work, can occupy a larger relative share in smaller models because the vocabulary size is often kept fixed while the transformer body is scaled down. For a dense unembedding layer with vocabulary size v and hidden size d, the output projection contains vd weights, or reuses a tied embedding matrix of this size, and computes one score for each vocabulary token. The Qwen3-0.6B draft model used in this thesis has $v = 151{,}936$ and $d = 1024$ [8], [9], giving

$$vd = 151{,}936 \times 1024 = 155{,}582{,}464$$

projection entries. Relative to the nominal 0.6B model scale, this matrix size corresponds to approximately 25.9% of the parameter budget. By contrast, the Qwen3-32B-AWQ target model uses $d = 5120$ with the same vocabulary size [10], so its unembedding matrix has 777,912,320 entries, which is only about 2.4% of the nominal 32B scale. For models with tied embeddings, this should be read as the size of the matrix used by the unembedding layer rather than as an additional untied parameter block. Although parameter share is not identical to runtime share,

the calculation illustrates why the full-vocabulary projection can become especially important for small draft models.

The main inference metric in this thesis is decode throughput, measured as generated tokens per second. Throughput directly describes how quickly a system emits output tokens after the prompt has been processed, and it is the primary metric used to compare baseline decoding, dense-draft speculative decoding, and FlashHead-draft speculative decoding. Time to first token (TTFT) and time per output token (TPOT) are complementary latency metrics, but the central comparison in this work is based on output tokens per second.

Different inference optimizations target different parts of this pipeline. KV-cache management and paged attention improve memory utilization during serving [2]. FlashAttention reduces attention memory traffic and improves GPU kernel efficiency [3], [4]. Quantization reduces weight bandwidth and arithmetic cost, and is particularly relevant to AWQ-style deployments [11]. Speculative decoding reduces the number of expensive target-model decode steps by using a smaller draft model to propose several tokens at once [5], [6]. Finally, ANN-style unembedding-layer replacements target the vocabulary projection itself by avoiding a full dense score over all tokens at every draft step.

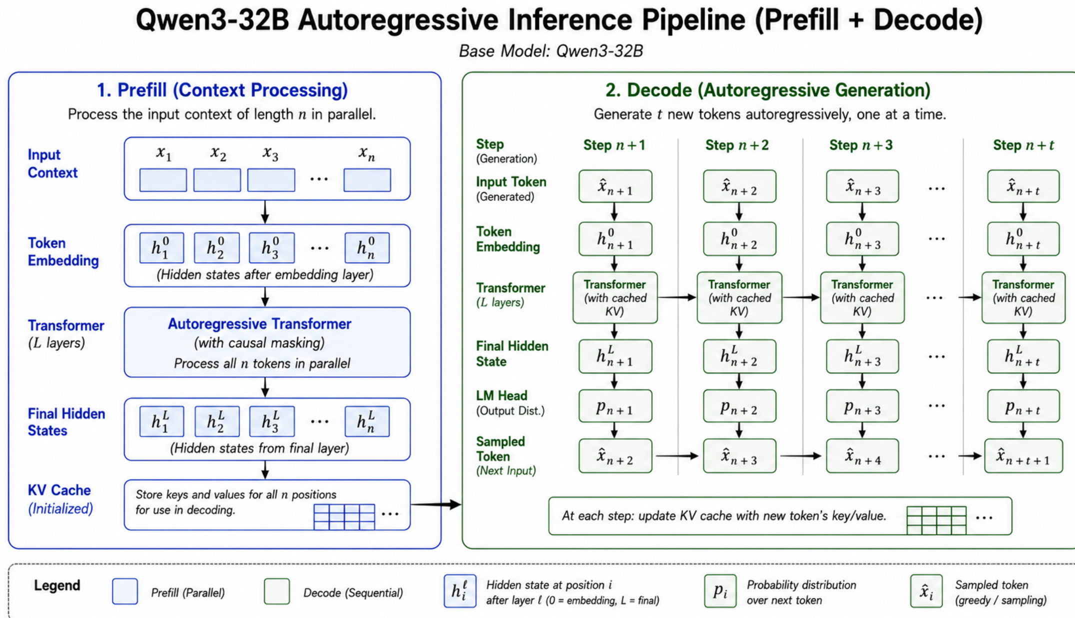

Figure 2.1: Autoregressive decoding flow for the Qwen3-32B target model.

2.2 Speculative Decoding

Speculative decoding accelerates autoregressive generation by separating token proposal from token verification. Instead of asking a large target model to produce every token sequentially, a cheaper draft model proposes a short block of candidate tokens, and the target model verifies this block in one forward pass [5], [12]. Figure 2.2

illustrates this proposal-and-verification loop.

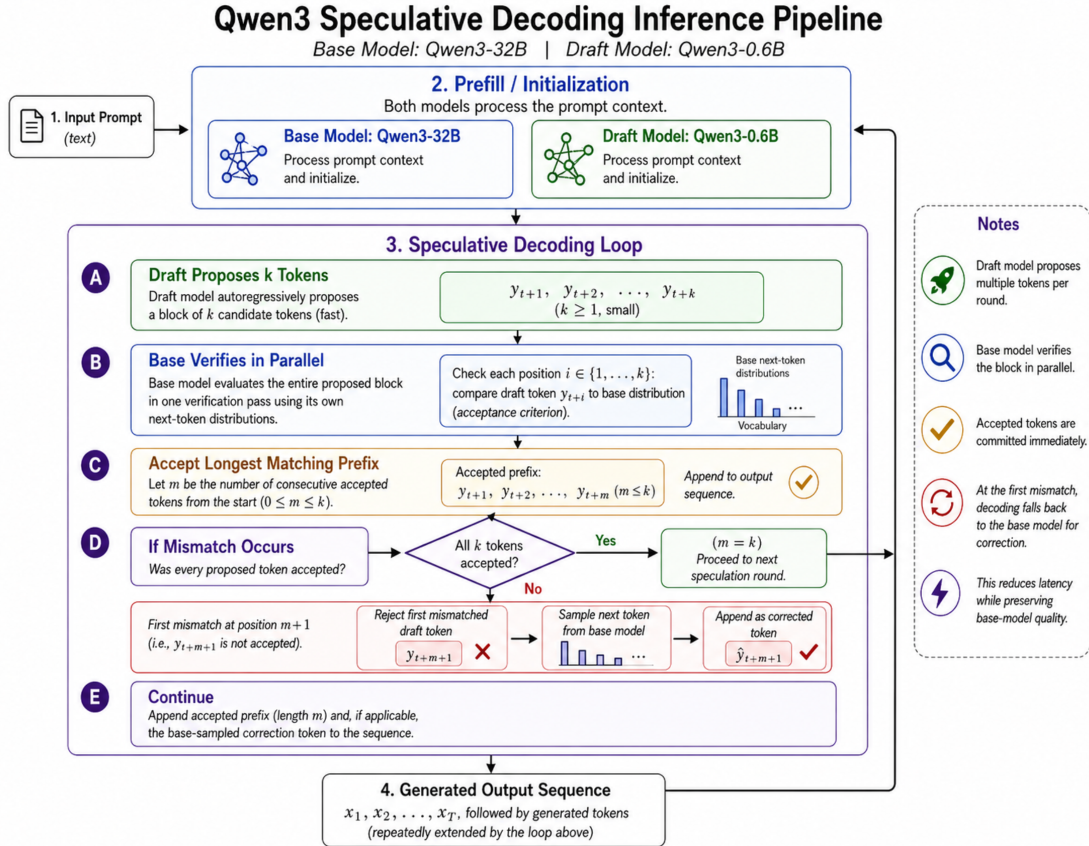

Figure 2.2: Speculative decoding with a draft model that proposes candidate tokens and a target model that verifies the proposed continuation.

Let p denote the target-model distribution and q the draft-model distribution. Starting from a prefix $x_{<t}$, the draft model proposes a block

$$y_{1:K} = (y_1, \ldots, y_K),$$

where K is the speculative block length. The target model is evaluated on the prefix extended by this draft block, producing logits for the proposed positions in parallel. If the draft tokens agree with the target model, several output tokens can be accepted after a single target-model call.

For stochastic decoding, speculative sampling uses a rejection-sampling rule that preserves the target-model distribution. At proposed position i, the draft token y_i is accepted with probability

$$\alpha_i = \min\left(1, \frac{p(y_i \mid x_{<t}, y_{<i})}{q(y_i \mid x_{<t}, y_{<i})}\right).$$

If a token is rejected, the replacement is sampled from the positive residual between the target and draft distributions [5], [12]. In the deterministic setting used in this thesis, the draft model proposes greedily and the target model verifies greedily. The accepted part is the longest prefix of draft tokens that matches the target model's own continuation; when a mismatch occurs, the target model supplies the correction token. The generated sequence is therefore the same as target-model greedy decoding.

The speedup depends on the balance between proposal cost and acceptance behavior. If $T_p(1)$ is the latency of one normal target-model decode step, $T_p(K)$ is the latency of verifying a block of length K, T_q is the latency of one draft step, and T_o represents cache and control overhead, the approximate speculative time per emitted token is

$$\mathrm{TPOT}_{\mathrm{spec}} \approx \frac{T_p(K) + KT_q + T_o}{\mathbb{E}[R]},$$

where R is the number of output tokens emitted by one speculative round. The method is beneficial when the draft model is cheap, block verification is cheaper than K independent target steps, and $\mathbb{E}[R]$ is large because the draft model often agrees with the target model [6].

2.3 Dense Unembedding Layer

At the end of each decoding step, the transformer body produces a hidden state $h_t \in \mathbb{R}^d$. The dense unembedding layer maps this hidden state to one logit per vocabulary token. With vocabulary size v and output embedding matrix $E \in \mathbb{R}^{v \times d}$, the dense layer computes

$$z_t = Eh_t + b,$$

where $z_t \in \mathbb{R}^v$ is the logit vector and $b \in \mathbb{R}^v$ is an optional bias term. Many modern language models tie the input embedding matrix and the output projection matrix, which reduces stored parameters but does not remove the need to multiply the hidden state by the full vocabulary matrix at inference time [13].

The logits define the next-token distribution through

$$p(x_t = i \mid x_{<t}) = \frac{\exp(z_{t,i}/\tau)}{\sum_{j=1}^{v} \exp(z_{t,j}/\tau)},$$

where τ is the sampling temperature. Greedy decoding corresponds to selecting $\arg\max_i z_{t,i}$, while sampling draws from the softmax distribution or from a truncated version of it, such as top-k or nucleus sampling.

The dense unembedding layer performs $\Theta(vd)$ dot-product work per token because it computes a dot product between h_t and every row of E. It also reads $\Theta(vd)$ weights and writes $\Theta(v)$ logits. For a batch of B hidden states, the operation becomes a

matrix multiplication with arithmetic work $\Theta(Bvd)$, but autoregressive decoding often uses small effective batch sizes, especially in low-latency or interactive settings. In that regime the dense unembedding layer can be limited not only by arithmetic throughput but also by memory bandwidth, since the full output matrix must be scanned at every generated token.

In speculative decoding this cost is paid on the draft side for every proposed token. A block of K draft tokens therefore performs K dense unembedding evaluations, giving draft-unembedding work of $\Theta(Kvd)$ per speculative round before considering softmax, sampling, and synchronization overheads. This is the part of the draft model that FlashHead targets.

2.4 Approximate Nearest Neighbor Search

Nearest neighbor search asks for the database vector most similar to a query vector under a chosen similarity measure, such as inner product, cosine similarity, or Euclidean distance. Exact search over a large database requires comparing the query with every stored vector. In an unembedding layer, this is analogous to comparing the hidden state h_t with every vocabulary embedding. Approximate nearest neighbor (ANN) search relaxes exactness: it tries to retrieve vectors that are close to the true nearest neighbors while reducing search time, memory traffic, or both [14], [15].

Most ANN methods use an index that organizes vectors before inference. Locality-sensitive hashing maps similar vectors to the same buckets with high probability, clustering and inverted-file methods search only selected regions of the vector space, and graph-based methods traverse a sparse neighborhood graph. These approaches differ in implementation, but they share the same basic trade-off: probing more buckets, clusters, or graph nodes improves recall, while probing fewer candidates reduces computation. Multi-probe search makes this trade-off explicit by checking several likely regions of an index rather than only one [16].

For language-model inference, the database vectors are the rows of the output embedding matrix and the query is the current hidden state. An ANN-style unembedding replacement can first retrieve a small candidate set of likely vocabulary tokens, then compute exact logits only for those candidates. This changes the dense full-vocabulary ranking problem into a retrieval-and-reranking problem. The approximation must still preserve enough top-token accuracy for the decoding method being used; in speculative decoding, a cheaper draft unembedding layer is useful only if it does not reduce target-model acceptance enough to cancel the latency saving.

2.5 FlashHead

FlashHead is a training-free, drop-in replacement for the dense unembedding layer that reframes vocabulary projection as a retrieval problem [7]. A dense unembedding layer ranks tokens by computing inner products $e_i^\top h_t$ against all vocabulary embeddings, while FlashHead first retrieves promising token clusters and then scores

only the tokens inside those clusters. Figure 2.3 contrasts the dense full-vocabulary projection with the FlashHead two-stage retrieval path.

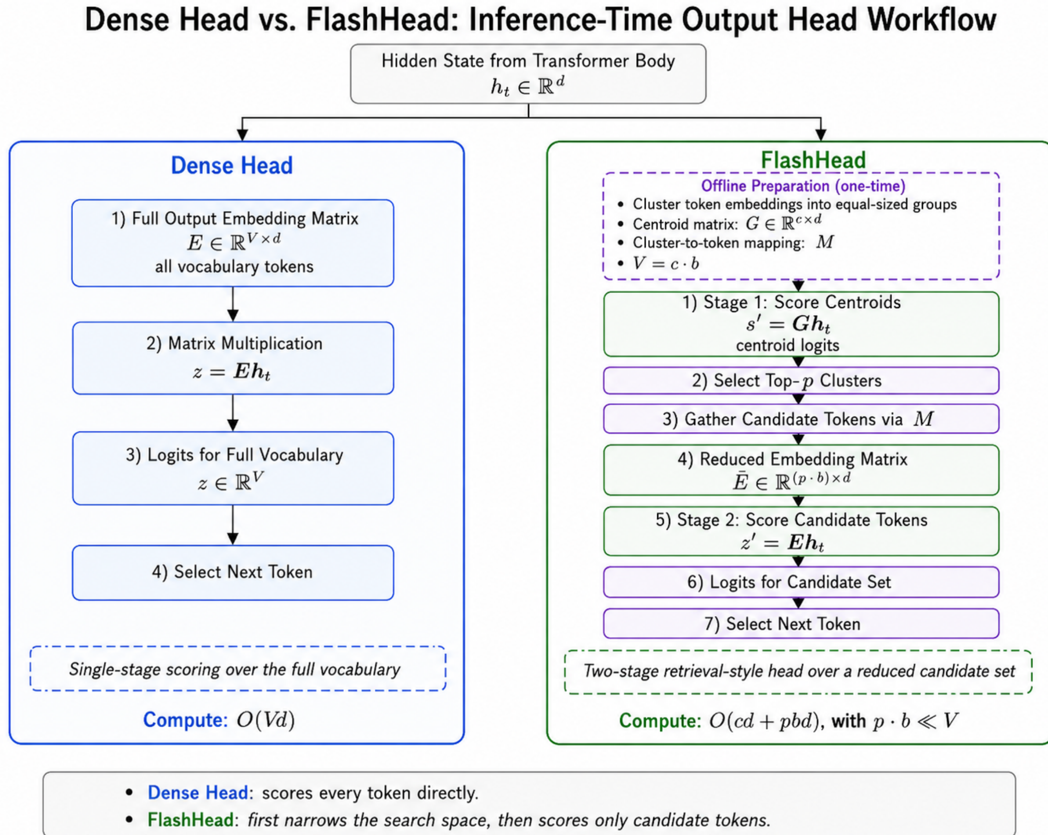

Figure 2.3: Dense unembedding scoring compared with FlashHead's cluster-first retrieval and reduced candidate scoring.

The method begins with an offline clustering step. The rows of the output embedding matrix E are partitioned into c equal-sized clusters using a spherical k-means style objective [17]. If $b = v/c$ is the cluster size, FlashHead stores a centroid matrix

$$G \in \mathbb{R}^{c \times d}$$

and a cluster-to-token index matrix

$$M \in \{1, \ldots, v\}^{c \times b}.$$

Here $M_{j,\ell}$ stores the vocabulary index of the ℓth token assigned to cluster j. The equal-size constraint is important because it makes M a dense rectangular tensor rather than a ragged structure. Candidate token indices can therefore be gathered with predictable memory access, which is one of the main hardware-oriented differences between FlashHead and a generic inverted-file approximate-nearest-neighbor index.

At inference time, FlashHead first scores all centroids:

$$s_t = Gh_t,$$

where $s_t \in \mathbb{R}^c$ contains one score per cluster. For greedy decoding, the method selects the top p centroids. For sampling, FlashHead can sample p centroids without replacement from the centroid-level distribution, which extends the retrieval method beyond only top-token search [7]. This multi-probe design is inspired by retrieval systems that probe several likely regions of an index rather than a single bucket [16].

Let P_t be the selected set of clusters. FlashHead gathers the candidate token set and scores only those embeddings:

$$S_t = \{M_{j,\ell} \mid j \in P_t,\ \ell = 1, \dots, b\}, \qquad \tilde{z}_t = E_{S_t} h_t,$$

where $|S_t| = pb$ when the selected clusters are distinct. The idealized dot-product work changes from dense unembedding work

$$\Theta(vd)$$

to two-stage FlashHead work

$$\Theta(cd) + \Theta(pbd) = \Theta\left(\left(c + \frac{pv}{c}\right) d\right).$$

Ignoring gather overhead and kernel effects, the relative dot-product work is

$$\rho = \frac{c + pb}{v} = \frac{c}{v} + \frac{p}{c}.$$

This expression shows the main trade-off: more clusters increase centroid cost but reduce cluster size, while more probes improve coverage but increase second-stage token scoring. In the Qwen3-0.6B-FlashHead checkpoint used in this thesis, the stored clustering has $v = 151{,}936$ tokens and $c = 9496$ clusters, so $b = 16$ tokens per cluster. FlashHead can also quantize the first stage because the centroid matrix is static and the second stage recomputes token logits on the selected candidates [7]. In this thesis, FlashHead is used only as the draft-model unembedding layer; the target model and verification rule remain unchanged, so end-to-end gains depend on reducing draft-side unembedding latency while preserving a high speculative acceptance rate.

3

Methods

3.1 Overview

The method evaluates a draft-head replacement inside speculative decoding. The base model and the speculative verification procedure are kept fixed. The dense draft model and the FlashHead draft model use the same model scale, but differ in the head used to produce draft tokens.

The final comparison contains three decoding modes:

1. **Baseline decoding**: the Qwen3-32B-AWQ base model generates tokens without speculative decoding.
2. **Dense draft speculative decoding**: the Qwen3-0.6B draft model proposes candidate tokens using its dense LM head.
3. **FlashHead draft speculative decoding**: the Qwen3-0.6B-FlashHead model proposes candidate tokens using the existing FlashHead head.

3.2 Models

The base model in the final experimental scope is Qwen3-32B-AWQ. The dense draft model is Qwen3-0.6B. The FlashHead draft model is an existing Qwen3-0.6B-FlashHead checkpoint.

The FlashHead checkpoint contains a prebuilt vocabulary clustering structure. In the local model assets, the clustering configuration records a vocabulary size of 151,936, hidden size 1024, and 9496 clusters. These values are used as implementation details of the existing FlashHead model, not as a newly trained contribution of this thesis.

3.3 Speculative Decoding Implementation

The implemented decoding protocol is deterministic greedy speculative decoding. The target model and draft model each maintain their own generation state, including the attention cache, the attention mask, the current prefix length, and the model's current greedy next-token prediction. In the baseline condition, only the target model is used and generation proceeds autoregressively one token at a time. In the speculative condition, the target model first performs the prompt prefill and emits the first output token. This first token is then appended to both the target and draft

states so that the two KV caches represent the same prefix before speculative rounds begin.

Each speculative round has four conceptual stages:

1. **Draft proposal**: starting from the shared prefix, the draft model greedily proposes at most K tokens, where K is the configured speculative block length.
2. **Target verification**: the target model checks whether its own greedy continuation agrees with the proposed block.
3. **Emission**: the accepted prefix of the proposal is written to the output sequence. If the first mismatch occurs before the end of the block, the target model's correction token is emitted after the accepted draft tokens.
4. **State alignment**: both model states are advanced to the emitted sequence, and any speculative draft-cache positions beyond the accepted prefix are discarded.

The verifier exploits the fact that greedy decoding only needs agreement with the target argmax. Before running an expensive target-model verification over the whole block, the implementation compares the target model's already available next-token prediction with the first draft proposal. If they differ, the accepted draft length is zero and the verifier can skip the batched target forward pass for that block. If they agree, the target model evaluates the proposed continuation in one batched forward pass, and the implementation compares the target argmax at each position with the corresponding draft token. The accepted draft length is the longest matching prefix. This procedure preserves the exact output of target-model greedy decoding: accepted tokens are those the target model would also have selected, and every rejection is corrected by the target model's own next token.

3.4 FlashHead Integration

FlashHead is integrated as a separate draft-side token-selection component rather than as a replacement for the whole draft model. At a high level, the draft model first computes the final hidden state in the usual way. FlashHead is then attached at the output-projection stage, where that hidden state is converted into next-token scores. The FlashHead component is constructed from the checkpoint assets that describe the vocabulary clustering. These assets include the cluster centroids, the mapping from clusters to vocabulary tokens, and the special-token configuration needed to make the component compatible with the tokenizer and output embedding matrix.

In the dense draft condition, draft-token proposal uses the ordinary LM head of the draft checkpoint. In the FlashHead condition, the transformer body is still executed normally up to the final hidden state, but next-token selection is performed by FlashHead instead of by a full dense vocabulary projection. Thus, FlashHead changes only the draft-side token-selection mechanism. The target model, target verifier, speculative acceptance rule, prompt processing, and output emission rule are unchanged. This design isolates the method under study: the experiment asks

whether replacing the draft LM head with an ANN-style head reduces proposal cost enough to improve the whole speculative decoding pipeline.

A practical issue in the FlashHead path is synchronization. A direct next-token API may return a Python scalar or otherwise force the proposed token to be materialized on the CPU. Such a synchronization can dominate the apparent head latency when the actual GPU computation is small. For the optimized benchmark paths, the FlashHead retrieval and reduced candidate-scoring stages are therefore invoked in a no-synchronization form that keeps the selected token as a GPU tensor until the surrounding speculative loop requires a CPU-visible value. This does not change the mathematical token selected by FlashHead; it only removes avoidable host-device synchronization from the proposal path.

3.5 Diagnostic Profiling

Before interpreting throughput results, a small diagnostic profiling experiment is used to identify where time is spent inside the implemented speculative loop. This profiling run is not treated as the final benchmark. Its purpose is to decompose the end-to-end runtime into draft-head cost, draft-transformer cost, target verification, finalization, cache alignment, and CPU–GPU synchronization.

The profile compares three draft configurations:

- **Dense draft**: a standard dense draft checkpoint using its ordinary LM head.
- **Dense shared**: the transformer body from a FlashHead checkpoint, but with dense LM-head token selection.
- **FlashHead draft**: the same FlashHead checkpoint body with FlashHead token selection.

The dense-shared condition is an important control. It separates effects caused by a different checkpoint or transformer body from effects caused by the head implementation itself. Without this control, a difference between dense and FlashHead runs could be incorrectly attributed to the head even if it originated from the underlying draft checkpoint.

The profiler records wall-clock phase shares for the initial target step, draft proposal, target verification, and finalization. It also decomposes the draft side into prefill, proposal, cache alignment, transformer-body time, head time, scalar materialization, and explicit synchronization. Verification is analyzed not only by total time but also by behavior: how often verification is skipped because the first draft token already disagrees with the target, how often no draft tokens are accepted, how often a whole block is accepted, and how accepted lengths are distributed across speculative rounds. The profile additionally estimates a head-only ceiling, which indicates the maximum possible gain that could be obtained if only the draft-head component were accelerated and all other costs remained unchanged.

The diagnostic configuration intentionally uses a small number of prompts and iterations. The default profile uses two GPUs, with the target model on the first

GPU and the draft model on the second, one tensor-parallel shard per model, a maximum of 64 generated tokens, a speculative block length of $K = 6$, one warmup iteration, one measured iteration, and two representative prompts from the fixed diagnostic subset. The profiling instrumentation introduces additional CUDA events and synchronization points, so its absolute tokens-per-second values should not be interpreted as final performance. The meaningful quantities are the relative phase shares and comparisons between modes within the same profiled run.

3.6 Optimization of the Speculative Loop

The profiling results motivate optimizations beyond simply replacing the dense draft head. In a native speculative implementation, each round contains several operations that can reduce the benefit of a faster draft head: the target model may need a single-token append after verification, the draft model may need to be realigned before the next proposal, and Python scalar extraction can synchronize the CPU with the GPU. The optimized implementation therefore studies how much overhead can be removed while preserving the same greedy speculative decoding semantics.

3.6.1 Pipelined finalization and proposal

The first optimization overlaps work across the two GPUs. In the ordinary schedule, after target verification the target model appends the final emitted token, the draft model appends the same token, and only then can the draft model begin proposing the next block. When the target and draft models reside on different GPUs, the target-side finalization and the draft-side preparation for the next block can partially overlap. The optimized schedule launches the target append asynchronously while the main thread advances the draft state and starts preparing the next proposals.

This pipelining can reduce the serial cost of a round from approximately

$$T_{\text{target finalize}} + T_{\text{draft prepare}} \tag{3.1}$$

to approximately

$$\max(T_{\text{target finalize}}, T_{\text{draft prepare}}), \tag{3.2}$$

when the hardware and runtime provide enough concurrency. The possible gain is therefore bounded by the smaller of the two overlapped components. This optimization cannot remove the target verification step itself, and its effectiveness depends on separate devices, CUDA stream behavior, and Python thread scheduling.

3.6.2 Deferred final-token verification

The most important optimization is deferred final-token verification. In the ordinary loop, a block that ends with a target correction or with the target token following a

fully accepted block requires an additional single-token target forward to append this final token to the target cache. This operation is algorithmically necessary in the sense that the target cache must eventually contain the emitted token, but it does not have to be performed as a separate target-model call immediately after the block.

The deferred schedule stores the final target token from the current round and prepends it to the verifier input of the next round. In the next verification call, the target model therefore advances over the stored final token and the new draft proposals in one batched pass. On the draft side, the stored final token is appended before producing the next proposal block, so the draft still proposes from the correct emitted prefix. The key accounting detail is that the stored token was already emitted by the previous round. When it is included in the next verifier input, it is used to update the target cache and provide the context for checking the following draft tokens, but it is not counted again as a newly accepted draft proposal. Figure 3.1 illustrates this deferred flow.

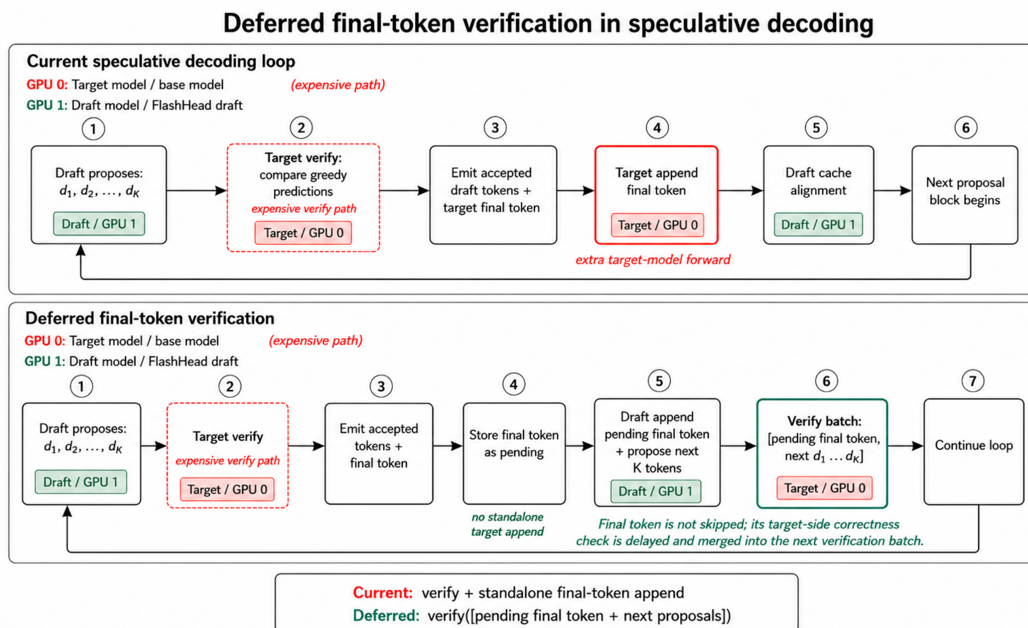

Figure 3.1: Deferred final-token verification. The final target token emitted in one speculative round is carried into the next target verification batch, allowing the target cache to advance without a separate single-token target forward for most rounds.

The optimization does not remove target-model computation for the final token. Instead, it changes where that computation occurs. Let I be the initial prompt and first-token cost, D_i the draft proposal cost in round i, V_i the target verification cost, and F_i the standalone finalization cost. A simplified ordinary schedule can be written as

$$T_{\text{ordinary}} = I + \sum_i (D_i + V_i + F_i). \tag{3.3}$$

With deferred final-token verification, most standalone target finalization calls are folded into the following verification call:

$$T_{\text{deferred}} \approx I + D_1 + \sum_i (V_i' + D_i') + T_{\text{tail}}. \tag{3.4}$$

Here V_i' may be slightly larger than V_i because it includes the stored final token, and D_i' may differ because the draft side prepares from the updated prefix. The expected saving is the removed standalone finalization overhead minus the additional cost of the longer verification batches, extra draft preparation, and the final tail step at the end of generation. The mechanism is therefore most useful when many small target finalization calls are expensive relative to the marginal cost of verifying one additional token in a batched target forward pass.

The optimization experiments include target-only greedy decoding, current dense and FlashHead speculative decoding, dense and FlashHead deferred variants.

3.7 Benchmark Protocol

All experiments are launched through the shared benchmark framework, which provides model loading, prompt loading, warmup control, measurement loops, Slurm entry points, and result serialization. Common run parameters include numerical precision, tensor-parallel size, maximum generated tokens, warmup iterations, measurement iterations, prompt file, output directory, and run name. Warmup iterations are excluded from reported measurements. The measured sample set is the Cartesian product of prompts and measurement iterations. For multi-mode profiling and optimization runs, the execution order of modes is alternated to reduce bias from a fixed ordering.

The main native speculative experiments use Qwen3-32B-AWQ as the target model. Dense draft runs use Qwen3 draft checkpoints, while FlashHead draft runs use corresponding FlashHead checkpoints with the clustering assets described above. Diagnostic and optimization runs typically place the target model on one GPU and the draft model on another with one tensor-parallel shard per model. This placement makes device overlap, finalization cost, proposal cost, and synchronization overhead visible. The more general speculative benchmark path also supports tensor-parallel target loading for larger deployments.

Prompt files are fixed subsets derived from the benchmark pool described in Section 4.1. The benchmark path uses flattened prompts for controlled latency and throughput measurement.

The protocol follows five reproducibility principles. First, decoding is greedy so that output differences are attributable to implementation changes rather than sampling

noise. Second, fixed prompt files are used for comparable runs. Third, warmup and measurement are separated to avoid mixing cold-start behavior with steady-state timing. Fourth, exact output directories are used so that result files are not silently overwritten. Fifth, each result directory stores the run configuration together with raw outputs and summaries, allowing reported tables to be traced back to model names, precision, devices, prompt sets, speculative block length, warmup iterations, and measurement iterations.

4

Experimental Results

This chapter reports the measured performance results. It first describes the benchmark dataset used for the experiments and defines the evaluation metrics. The discussion then starts from the isolated 0.6B draft model and its output head, moves to end-to-end speculative decoding, profiling evidence, and the final optimized FlashHead schedule. The purpose is to distinguish local head-level acceleration from system-level throughput improvement.

4.1 Benchmark Dataset

The evaluation prompts are drawn from Spec-Bench, a benchmark and unified evaluation platform for speculative decoding methods [6]. Spec-Bench is designed to compare speculative decoding approaches under shared test conditions rather than under method-specific setups. Its public repository provides the benchmark question file used in this thesis. The benchmark contains 480 examples organized as six 80-example subtasks: multi-turn conversation from MT-Bench, retrieval-augmented generation and question answering based on Natural Questions, summarization from CNN/Daily Mail, German-to-English translation from WMT14, and mathematical reasoning from GSM8K. In this work, fixed subsets of this prompt pool are used for profiling and optimization sweeps, while larger or full subsets are reserved for broader benchmark coverage.

4.2 Evaluation Metrics

The evaluation separates prompt processing from token-by-token decoding. Time to first token (TTFT) covers the target-model prompt prefill and first-token generation. Decode time excludes this prefill part and measures the period after the first emitted token. Decode throughput is therefore computed as the number of decode-phase output tokens divided by decode-phase wall time. Acceptance is measured as the fraction of proposed draft tokens that are accepted by the target verifier. For performance analysis, the implementation also records phase-level time for the initial step, draft proposal, target verification, finalization and cache alignment, and the FlashHead-specific draft-head path. Target verification is measured both as end-to-end wall time and, where CUDA events are available, as the GPU time of the target forward pass itself. The distinction matters because wall time also includes Python scheduling, tensor bookkeeping, and possible synchronization overhead.

4.3 Model-only Results for the 0.6B Draft Model

The first result isolates the 0.6B draft model from the target model. Two levels are considered. The first measures only the output head from a fixed hidden state to next-token prediction. The second measures a complete single-token decode step of the draft model, including the transformer body and the output head.

4.3.1 Dense LM Head vs. FlashHead

The LM-head comparison uses single-step head latency. Under the same hidden-state input, only the time from hidden state to next-token prediction is measured for the dense LM head and FlashHead. The primary metric is mean latency, while P50 latency, P95 latency, and token throughput are reported as supporting statistics. Table 4.1 reports the latency measurements, and Table 4.2 summarizes the corresponding speedup.

Table 4.1: Single-step head latency of the dense LM head and FlashHead.

Mode	Mean latency (ms)	P50 (ms)	P95 (ms)	Tok/s
Dense LM head	2.493	2.494	2.500	401.199
FlashHead	0.577	0.575	0.593	1733.562

Table 4.2: Head-level speedup from the dense LM head to FlashHead.

Comparison	Metric	Dense head	FlashHead	Speedup
Dense LM head vs. FlashHead	Single-step mean latency	2.493	0.577	4.321×

FlashHead provides a strong isolated head-level acceleration. The mean head latency decreases from 2.493 ms to 0.577 ms, giving a 4.321× speedup. The P50 and P95 values follow the same pattern, indicating that the reduction is not caused by a small number of outlier measurements.

This result establishes the local effect of replacing the dense projection with FlashHead. The next subsection measures whether this saving remains visible when the full draft-model decode step, including the unchanged transformer body, is included.

4.3.2 0.6B Dense Model vs. 0.6B Flash Model

Table 4.3 reports complete single-step decode latency for the dense and Flash models, and Table 4.4 gives the resulting model-level speedup.

Table 4.3: Complete single-step decode latency of the 0.6B dense and Flash models.

Mode	Mean latency (ms)	P50 (ms)	P95 (ms)	Tok/s
Dense model	18.330	18.308	18.508	54.554
Flash model	16.550	16.539	16.707	60.422

Table 4.4: Complete model-level speedup from the dense model to the Flash model.

Comparison	Metric	Dense	Flash	Speedup
Dense model vs. Flash model	Mean decode-step latency	18.330	16.550	1.108×

At the complete model level, the Flash model reduces the mean single-step decode latency from 18.330 ms to 16.550 ms. This corresponds to a speedup of 1.108×, or approximately 1.1×. The improvement shows that the FlashHead path remains visible even when the full draft-model decode step is measured.

Compared with the 4.321× head-level speedup, the complete draft-model speedup is much smaller. A complete decode step includes not only the LM head, but also the transformer body, KV-cache access, scheduling overhead, and synchronization. These components are unchanged by FlashHead and therefore dilute the benefit of accelerating only the token-selection head.

4.4 Speculative Decoding with FlashHead

The next experiment evaluates whether the local acceleration of FlashHead improves end-to-end speculative decoding. The baseline is target-model decoding without a draft model. The two speculative modes use the same target model, but differ in whether the 0.6B draft model uses the dense LM head or FlashHead. Table 4.5 reports the end-to-end measurements, and Table 4.6 summarizes the decode-throughput speedups.

Table 4.5: End-to-end speculative decoding results with dense and FlashHead draft models.

Mode	Decode Tok/s	TTFT (ms)	Acceptance	Decode total (ms)	Draft propose (ms)	Base verify (ms)	Finalize (ms)
Baseline	5.715	1156.704	–	44646.485	–	–	–
Dense draft	9.769	1166.982	0.437	27301.491	6001.347	8783.647	12352.848
FlashHead draft	9.823	1155.533	0.430	27148.480	5657.390	8823.664	12502.015

Table 4.6: Decode-throughput speedups in speculative decoding.

Comparison	Metric	A	B	Speedup
Baseline vs. dense draft	Decode Tok/s	5.715	9.769	1.709×
Baseline vs. FlashHead draft	Decode Tok/s	5.715	9.823	1.719×
Dense draft vs. FlashHead draft	Decode Tok/s	9.769	9.823	1.006×

Both speculative modes are substantially faster than baseline decoding. The dense draft improves decode throughput from 5.715 tok/s to 9.769 tok/s, a 1.709× speedup. The FlashHead draft reaches 9.823 tok/s, corresponding to a 1.719× speedup over baseline.

The difference between the dense and FlashHead draft modes is much smaller. FlashHead reduces draft proposal time from 6001.347 ms to 5657.390 ms, but end-to-end decode throughput increases only from 9.769 tok/s to 9.823 tok/s, or 1.006×.

The acceptance rate is also slightly lower for the FlashHead draft, decreasing from 0.437 to 0.430.

These results show that FlashHead does reduce the draft-side proposal path, but this local saving is mostly diluted in the complete speculative decoding loop. Base-model verification and finalization account for a large portion of the total runtime, so a faster draft head alone does not translate into a proportional throughput improvement.

4.5 Profiling Results and Implications

The profiling experiment decomposes speculative decoding into phase-level costs. It is used to explain why a large head-level speedup produces only a small end-to-end improvement in the previous experiment. Table 4.7 reports the measured phase shares, and Table 4.8 summarizes the profiling diagnosis.

Table 4.7: Phase-level profile of dense, dense-shared, and FlashHead draft modes.

Mode	Decode Tok/s	Decode total (ms)	Draft propose share	Base verify share	Finalize share	Acceptance	Mean emitted
Dense draft	7.587	16606.893	18.9%	32.7%	45.4%	0.333	2.930
Dense shared	7.643	16486.728	19.0%	33.1%	45.8%	0.333	2.930
FlashHead draft	7.776	16203.252	17.8%	33.5%	46.4%	0.333	2.930

Table 4.8: Profiling diagnosis and supporting evidence.

Diagnosis	Evidence
Base verification and finalization dominate	In the FlashHead draft mode, 79.9% of measured time is outside draft proposal.
The head-only ceiling is low	The proposal head accounts for only about 0.5% of decode wall time.
The draft body dominates proposal	The draft transformer body accounts for about 96.0% of proposal wall time, while the head accounts for about 2.7%.
Acceptance is low	Acceptance is 0.333 and the mean number of emitted tokens per round is 2.930.

The profiling results confirm that replacing only the head is insufficient for a large end-to-end gain. The FlashHead draft mode improves throughput from 7.587 tok/s to 7.776 tok/s compared with the dense draft, and its draft-proposal share decreases from 18.9% to 17.8%. Nevertheless, the dominant costs remain outside the proposal path: base verification and finalization together account for 79.9% of the FlashHead draft runtime.

The profile also shows why the isolated 4.321× head-level acceleration has a low system-level ceiling. The proposal head itself contributes only about 0.5% of total decode wall time, and even inside the proposal stage the draft transformer body dominates the runtime. Thus, accelerating the head alone cannot remove the main bottlenecks.

The practical implication is that FlashHead must be combined with scheduling changes in the speculative loop. The next optimization target is therefore not only faster token selection, but also fewer standalone target-model final-token append calls. One way to achieve this is to fold the final token into the next verification batch, so that the target cache is advanced without a separate append operation in most rounds.

4.6 Deferred Final-token Verification for Dense and FlashHead

The final result evaluates deferred final-token verification for both dense and FlashHead draft modes. Both variants use the same deferred final-token verification strategy, with the target model, prompt set, maximum generation length, speculative block length, and greedy decoding configuration kept fixed. This isolates the remaining difference more closely to the draft model and its output head. Table 4.9 reports the end-to-end comparison, Table 4.10 gives the decode-throughput speedups, and Table 4.11 reports the draft-path measurements.

Table 4.9: End-to-end comparison with deferred verification enabled for dense and FlashHead draft models.

Mode	Samples	Decode Tok/s	TTFT (ms)	Decode total (ms)	Acceptance	Mean emitted	Base verify calls	Base append calls
Base only	560	5.326	1204.115	6624634.302	–	–	0	35280
Dense deferred	560	10.731	1205.479	3287789.419	0.239	3.666	9407	560
FlashHead deferred	560	11.374	1200.069	3101688.131	0.237	3.642	9467	560

Table 4.10: Decode-throughput speedups with deferred verification.

Comparison	Metric	A	B	Speedup
Base only vs. dense deferred	Decode Tok/s	5.326	10.731	2.015×
Base only vs. FlashHead deferred	Decode Tok/s	5.326	11.374	2.136×
Dense deferred vs. FlashHead deferred	Decode Tok/s	10.731	11.374	1.060×

Table 4.11: Draft-path comparison under deferred verification.

Mode	Draft path (ms)	Draft path share	Draft ms/decode tok	Draft path Tok/s
Dense deferred	1438945.165	43.8%	40.786	24.518
FlashHead deferred	1289917.106	41.6%	36.562	27.351

With deferred verification enabled for both draft models, speculative decoding is substantially faster than base-only decoding. Dense deferred decoding improves throughput from 5.326 tok/s to 10.731 tok/s, corresponding to a 2.015× speedup over the base-only run. FlashHead deferred decoding reaches 11.374 tok/s, corresponding to a 2.136× speedup over base-only decoding.

The fair dense-versus-FlashHead comparison is therefore 10.731 tok/s against 11.374 tok/s, or a 1.060× end-to-end speedup. This is still far below the isolated FlashHead acceleration measured in Section 4.3.1, and also below the complete draft-model speedup in Section 4.3.2. Nevertheless, it is larger than the 1.006× improvement observed in the ordinary speculative comparison in Section 4.4. This indicates that the speculative-decoding pipeline does dilute the speedup provided by FlashHead. The throughput difference is not caused by better acceptance behavior. Acceptance is nearly unchanged, moving from 0.239 for dense deferred decoding to 0.237 for FlashHead deferred decoding, and the mean number of emitted tokens per round

changes from 3.666 to 3.642. Instead, the improvement comes from a lower draft-side cost. The combined draft path decreases from 1438945.165 ms to 1289917.106 ms, a 1.116× draft-path speedup, while draft-path time per decoded token decreases from 40.786 ms to 36.562 ms.

5

Conclusion

5.1 Summary

This thesis investigated whether an approximate-nearest-neighbor based draft head can improve large language model inference in a speculative decoding pipeline. The motivation was that small draft models still pay the cost of repeated full-vocabulary LM-head projection, and this cost can be relatively large compared with the size of the draft transformer. FlashHead was therefore studied as a practical ANN-style replacement for the dense draft LM head.

The work first introduced autoregressive decoding, speculative decoding, dense LM heads, and the FlashHead retrieval-based head structure. It then implemented a controlled comparison around Qwen3 models. The target model was kept fixed as Qwen3-32B-AWQ, while the draft side compared a Qwen3-0.6B dense draft model against a Qwen3-0.6B-FlashHead draft model. The speculative decoding rule, target verification logic, prompt processing, and output emission behavior were kept unchanged so that the effect of replacing the draft head could be isolated as much as possible.

The experiments were organized at several levels. First, the 0.6B draft model was evaluated outside speculative decoding to measure both complete single-step decode latency and isolated head latency. Second, dense-draft and FlashHead-draft speculative decoding were compared against target-only baseline decoding. Third, phase-level profiling was used to identify where time was spent inside the speculative loop. Finally, the work evaluated deferred final-token verification for both dense and FlashHead draft schedules, reducing the number of standalone target append operations while keeping the dense-versus-FlashHead comparison fair.

Across these experiments, the thesis showed that FlashHead is effective as a local head acceleration method. The isolated head latency decreased from 2.493 ms for the dense LM head to 0.577 ms for FlashHead, corresponding to a 4.321× speedup. At the complete draft-model step level, this became a smaller but still visible improvement from 18.330 ms to 16.550 ms, or 1.108×. These results confirm that the dense draft head can be accelerated, but also show that the head is only one part of the full inference path.

5.2 Conclusions

The main conclusion is that FlashHead is beneficial for reducing the local cost of draft-token selection, but this benefit does not automatically translate into a large end-

to-end speedup in speculative decoding. In the direct speculative comparison, both draft-based methods were much faster than target-only decoding. Dense speculative decoding improved throughput from 5.715 tok/s to 9.769 tok/s, and FlashHead speculative decoding reached 9.823 tok/s. However, the difference between dense draft and FlashHead draft was only 1.006× in decode throughput, even though the head alone was more than four times faster.

This gap is explained by the cost structure of the implemented speculative decoding pipeline. Profiling showed that base-model verification and finalization dominated the runtime. In the FlashHead draft profile, 79.9% of measured time was outside the draft proposal stage, while the proposal head itself accounted for only about 0.5% of total decode wall time. Even inside the proposal stage, the draft transformer body dominated the cost. Therefore, a faster head can reduce proposal latency, but the system-level gain is bounded when verification, cache alignment, target append calls, and Python or device synchronization remain expensive.

The results also show that speculative decoding should be optimized as a whole pipeline rather than as a collection of isolated components. With deferred final-token verification enabled for both draft heads, dense deferred decoding reached 10.731 tok/s and FlashHead deferred decoding reached 11.374 tok/s, giving a 1.060× end-to-end speedup. This remains far below the isolated head-level speedup, but it is larger than the 1.006× gain in the ordinary speculative comparison. The draft path itself improved by 1.116×, from 1438945.165 ms to 1289917.106 ms, showing that speculative decoding diluted the FlashHead gain but did not eliminate it.

Overall, the answer to the research questions is therefore mixed but informative. FlashHead clearly accelerates the draft head, and it can modestly reduce draft proposal cost. In a naive or ordinary speculative loop, however, this improvement is heavily diluted by target-side verification and finalization. The most promising direction is not simply to make the draft head faster in isolation, but to co-design FlashHead with a more efficient speculative decoding schedule. In this sense, FlashHead should be viewed as one useful component in a broader optimization of speculative decoding, rather than as a standalone solution for end-to-end inference acceleration.

5.3 Limitations and Future Work

Several limitations remain in the current work. The thesis uses an existing FlashHead checkpoint rather than training or reclustering a new FlashHead model for the exact experimental setting. The final evaluation focuses on Qwen3 models, especially a Qwen3-32B-AWQ target model and a Qwen3-0.6B draft model, so the conclusions should be validated on additional model families, hardware platforms, prompt types, batch sizes, and speculative block lengths. The experiments also focus on greedy decoding; future work should study whether the same trade-offs hold under stochastic sampling, where FlashHead's approximate retrieval behavior may interact differently with acceptance and output quality.

A natural next step is to make FlashHead adaptive. Instead of using a fixed retrieval budget for every token, an adaptive FlashHead policy could adjust the number of

probed clusters or candidate tokens according to the confidence of the centroid scores. For example, when the best cluster has a large margin over the next alternatives, FlashHead could use a smaller candidate set; when the margin is small, it could probe more clusters to protect top-token accuracy. This would reduce unnecessary computation on easy tokens while preserving quality on ambiguous tokens. Such an adaptive policy could also be connected to speculative decoding: tokens that are more likely to be rejected by the target model may not need the same retrieval budget as tokens whose agreement with the target is more likely.

Future work should also optimize speculative decoding itself. The profiling results suggest that verification and finalization are the main bottlenecks, so further work should reduce standalone target-model calls, fuse cache-update operations where possible, avoid CPU–GPU synchronization in the proposal path, and overlap draft-side preparation with target-side work. Adaptive speculative block lengths are another promising direction: the system could choose a larger K when recent acceptance is high and a smaller K when the draft model frequently disagrees with the target. This would help balance target-verification amortization against wasted draft computation.

Another direction is to co-design the draft model and FlashHead for acceptance rather than only for local head latency. The clustering structure, number of clusters, probe count, and candidate scoring strategy could be tuned for the target model used in verification. A draft head that is slightly more expensive but produces tokens with higher target agreement may outperform a cheaper head that causes more rejections. Finally, broader benchmarking should include memory overhead, multi-batch serving behavior, longer generations, and task-level quality checks. These extensions would clarify when FlashHead is beneficial, when it is neutral, and how it should be combined with speculative decoding optimizations to produce robust end-to-end speedups.

Bibliography

[1] S. Samsi et al., *From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference*, 2023. arXiv: 2310.03003.

[2] W. Kwon et al., *Efficient Memory Management for Large Language Model Serving with PagedAttention*, 2023. arXiv: 2309.06180.

[3] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*, 2022. arXiv: 2205.14135.

[4] T. Dao, *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*, 2023. arXiv: 2307.08691.

[5] Y. Leviathan, M. Kalman, and Y. Matias, *Fast Inference from Transformers via Speculative Decoding*, 2023. arXiv: 2211.17192.

[6] H. Xia et al., "Unlocking Efficiency in Large Language Model Inference: A Comprehensive Survey of Speculative Decoding," in *Findings of the Association for Computational Linguistics: ACL 2024*, Bangkok, Thailand: Association for Computational Linguistics, 2024, pp. 7655–7671. DOI: 10.18653/v1/2024.findings-acl.456 [Online]. Available: https://aclanthology.org/2024.findings-acl.456/

[7] W. Tranheden, S. Ahmed, D. Dubhashi, J. Matthiesen, and H. von Essen, "Flashhead: Efficient drop-in replacement for the classification head in language model inference," *arXiv preprint arXiv:2603.14591*, 2026.

[8] A. Yang, A. Li, B. Yang, et al., *Qwen3 Technical Report*, 2025. arXiv: 2505.09388.

[9] Qwen Team, *Qwen3-0.6B config.json*, 2025. Accessed: Apr. 28, 2026. [Online]. Available: https://huggingface.co/Qwen/Qwen3-0.6B/blob/main/config.json

[10] Qwen Team, *Qwen3-32B-AWQ config.json*, 2025. Accessed: Apr. 28, 2026. [Online]. Available: https://huggingface.co/Qwen/Qwen3-32B-AWQ/blob/main/config.json

[11] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*, 2024. arXiv: 2306.00978.

[12] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, *Accelerating Large Language Model Decoding with Speculative Sampling*, 2023. arXiv: 2302.01318.

[13] O. Press and L. Wolf, "Using the Output Embedding to Improve Language Models," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, Valencia, Spain: Association for Computational Linguistics, 2017, pp. 157–163. DOI: 10.18653/v1/E17-2025 [Online]. Available: https://aclanthology.org/E17-2025/

[14] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *Proceedings of the Thirtieth Annual ACM*

Symposium on Theory of Computing, Association for Computing Machinery, 1998, pp. 604–613. DOI: 10.1145/276698.276876

[15] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021. DOI: 10.1109/TBDATA.2019.2921572

[16] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB Endowment, 2007, pp. 950–961.

[17] I. S. Dhillon and D. S. Modha, "Concept Decompositions for Large Sparse Text Data Using Clustering," *Machine Learning*, vol. 42, no. 1–2, pp. 143–175, 2001. DOI: 10.1023/A:1007612920971

A

Experiment Artifacts

TODO: Add final run directories, Slurm scripts, and configuration files used for the thesis experiments.