SPECULATIVE DECODING · HOMELAB

Speculating on a Strix Halo

By Nate MacFadden · July 2026 · code private for now, happy to discuss

🤖 Human + AI (Claude) joint written 🤖

I saw DFlash at a very opportune time. I both was looking for some fun performance-focused project in ML, and I was looking for a way out of my growing software maintenance debt. That last bit was cryptic. I mean that I have an increasing number of software packages that I've developed for my Ph.D. work and I do my best to keep them stable but versions update, bugs are found, etc. To ease this work, I actually found LLMs to be a surprisingly well-suited tool: I can spin up an agent as a virtual user on-demand, have them run my code from scratch, and give me the detailed reviews/notes. To generate such reviews, I (well, in this case, Claude...) made a Claude Code plugin repo-review which simulates such users of different focuses (performance, correctness, documentation, ...). It makes the agents read the README, install the software, run tests, try to break it, etc. The agents measure performance so I want them to run on an uncontested computer, limiting me to sequential reviews/tests.

Since I want to run this review script a LOT, I ported it over to local models. In this way, I can have reviews running as a loop, hopefully catching issues that arise and reducing the number of bugs iteratively. As of when I wrote this, I was using a Qwen3.6-35B-A3B[1] on my Strix Halo box. The Strix does the thinking and a second machine (e.g., my laptop) does the doing: it installs the code, runs the tests, and calls over to the Strix whenever an agent needs a token. I picked that model since the Strix Halo performs better with MoE models and that one was said to be good, but the limiting factor in these reviews was (surprisingly) running this model, not the tests. There are a couple of obvious optimizations to the inference throughput like quantization and batching. Since I'm having reviews measure performance, batching of reviews would be so risky as to be avoided (with reviews in parallel on the same machine, performance numbers are contaminated); quantization is also out because it'd risk

further degrading the quality of the model. That could give a false sense of security on the code, which is arguably worse than no-review at all. (I come back to this in part 2: the quantized engine turns out to be so much faster that it changes the calculation.)

Speculative decoding

This leaves speculative decoding. I didn't really know what it was previously so I'm aiming to both somewhat introduce it here and record my experiments. I normally think of inference as a sequential process: one large model does a bit of work to generate a single token conditioned on the context, then repeats. The large model is large - this is what enables the high quality outputs but it also costs compute. Speculative decoding asks: what if we relegated the large model to just checking the results of a weaker draft model? That smaller model gets insight into the context (and, sometimes, the large model's activations), proposes a block of tokens, and then the big model (the "target") checks the entire draft in a single pass. This is really cool because it is a lossless[2] procedure: the output distribution is exactly the target's, at any temperature. Easiest to see under greedy decoding, where the target only accepts tokens it would have produced itself. In this way, you trade expensive target-model drafts for cheap drafter-model drafts and a single target-model verification pass. Thus we can speed up a single inference stream without loss of quality!

I first learned about this through DFlash[3]. This is a diffusion-style drafter that pushes the concept further. Instead of drafting tokens sequentially, it creates the entire draft in one pass via a small diffusion network. Coming from a performance-focused background, this sounds really attractive since the serial fraction is what caps your speedup (Amdahl), and this converts serial work into parallel. The details are that there is a 'block' of tokens (in this case, of size 16) in which the previous round's verify pass supplies 1 known token; the other 15 are filled in via the diffusion drafter.

I am/was a theorist so I wanted to understand the speedup η (defined in DFlash's paper). For γ drafted tokens per block, let τ be the average number of tokens committed per cycle: the accepted draft prefix plus the one token the target produces itself. In a standard autoregressive decoding, this would cost $\tau \cdot L_{target}$ time for L_{target} the latency. DFlash wins by replacing that with a computation costing $T_t + T_d$ time for T_t the time for the target model to verify and T_d the time for the draft model to draft, following from DFlash's Eq. 1:

$$\eta = \frac{\tau \cdot L_{target}}{T_t + T_d}$$

Generally $T_t \approx L_{target}$, but there are subtle differences here since I'm studying a MoE model (specifically, the verify pass here wakes more experts, so it's a bit slower). If I had $T_t = L_{target}$ exactly, then I'd achieve speedup as long as $T_d < (\tau - 1) L_{target}$. I.e., the quicker the draft is while still giving many good tokens τ, the better. That is the mental image to keep; the MoE wrinkle comes back below.

The right place to start is a baseline. z-lab released a drafter for my target, a 6-layer, 2048-wide network conditioned on 8 of the target's 40 hidden layers[4], so I measured the token rate for a Q8 quantized variant of the target as well as an unquantized version, the latter both with and without DFlash. This is an admittedly messy compari-

son since quantization and inference engine change simultaneously, but the PyTorch comparison cleanly shows the value of speculation:

ENGINE / PRECISION	SPECULATION	DECODE
PyTorch · ROCm · bf16	none	8.1 tok/s
PyTorch · ROCm · bf16	released DFlash drafter (tokens/cycle = $\tau \approx 4.9$)	**17.4 tok/s**
llama.cpp · Vulkan · Q8	none	47.4 tok/s

The 47.4 is llama-bench; the same build under llama-server measures 46.2. Both bf16 rows already include the fused-MoE kernel discussed further below, which the loader turns on by default.

On the bf16 side, the released drafter turns 8.1 tok/s into 17.4 tok/s, and it reproduced the target's tokens exactly (greedy decode) on every prompt I checked. The llama.cpp row is there for scale: quantization alone already beats the speculative PyTorch stack, and the obvious end-goal is all three at once, the fast engine, 8-bit weights, and speculation. That turns out to be surprisingly subtle even with DFlash recently merged[5] into llama.cpp mainline, and it is the subject of part 2. Part 1 stays in bf16 PyTorch for convenience.

Spoilers: what I found

- The draft is cheap next to the target ($T_d/T_t \approx 0.12$), but that cuts both ways: with $\eta = \tau \cdot L_{target}/(T_t + T_d)$, an infinitely fast drafter only takes the denominator from $1.12 \cdot T_t$ to T_t, a ~12% gain.
- A 16-token block verify costs only ~1.6× a single decode step *in the bf16 research stack*: the block wakes ~47 of 256 experts, but attention, embeddings, and head are read once for the whole block. Unfortunately, this costs ~4.2× on llama.cpp :(
- Across two seeds, early in training: among the four arms, the loss and the corruption schedule changed the ordering, and attention did not. The best loss was KL, which makes sense since what I'm doing here is analogous to distillation. The arm meant to isolate the loss was invalidated by an extraction mismatch (below), so how big the effect is stays open. If memory is tight, one can use a restricted vocabulary (like what Speculators[6] supports). By restricting to the top-32k most frequent tokens, one covers 97% of tokens used in held-out chats. This makes the drafter's output head 7.6× smaller but effectively no speedup, so only really use this for memory concerns.
- The target mixes full and linear attention, so keeping speculation lossless took a patch to the transformers cache path.

What the pieces cost

With the formula in hand, I measured the three timings on my own machine:

MEASURED ON THE RADEON 8060S (DRIFTS A FEW % RUN TO RUN)

one decode step: target, 1 token (L_{target}) **≈ 125 ms**

block verify: target, the 16-token block (T_t) **≈ 205 ms**

draft pass (T_d) **≈ 25 ms**

The ~47 experts per block quoted below were counted on the run that produced the τ of 4.9 and the 17.4 tok/s; later builds give closer to 48.5.

The first thing that jumps out is that the drafter is basically free: $T_d/T_t \approx 0.12$. So if a verify cost the same as one decode step, $\tau \approx 4.9$ would buy me ~4.1×.

It doesn't, though. The target checks all 16 positions in one pass, but each token routes to its own experts, so the block wakes ~47 of 256 instead of 8. Those 8, plus an always-on shared expert[1], are the expert share of the ~3B active. That puts the verify at $T_t/L_{target} \approx 1.6$... an unfortunate fact of life for MoE models. It could be worse. If the whole step scaled with the expert count, the verify would cost the naive 47/8 ≈ 6×, but attention, embeddings and the output head are read once for the whole block, so only the expert share grows.

Plugging it all in, theory says $\eta = \tau \cdot L_{target}/(T_d+T_t) = 4.9 \cdot 125\ ms/230\ ms \approx 2.7$, against the $2.1 = 17.4/8.1$ tok/s I actually measured. Close enough that I believe the model, with the gap probably being per-cycle overhead I haven't broken down yet. DFlash's own numbers agree[3], incidentally: that >6× headline is on a dense Qwen3-8B, and their Table 3 has Qwen3-Coder-30B-A3B at 2.6-3.5× at concurrency 1 despite τ of 6.4-8.1.

Draft the "right" way

So obviously I wanted to poke at this drafter myself. Before training anything, I went looking for the recipe everyone uses. There isn't one. DFlash's paper[3], their inference code[7], and Speculators[6] each pin down a recipe, and they disagree with each other. And where they *did* agree, I often couldn't find a stated reason why.

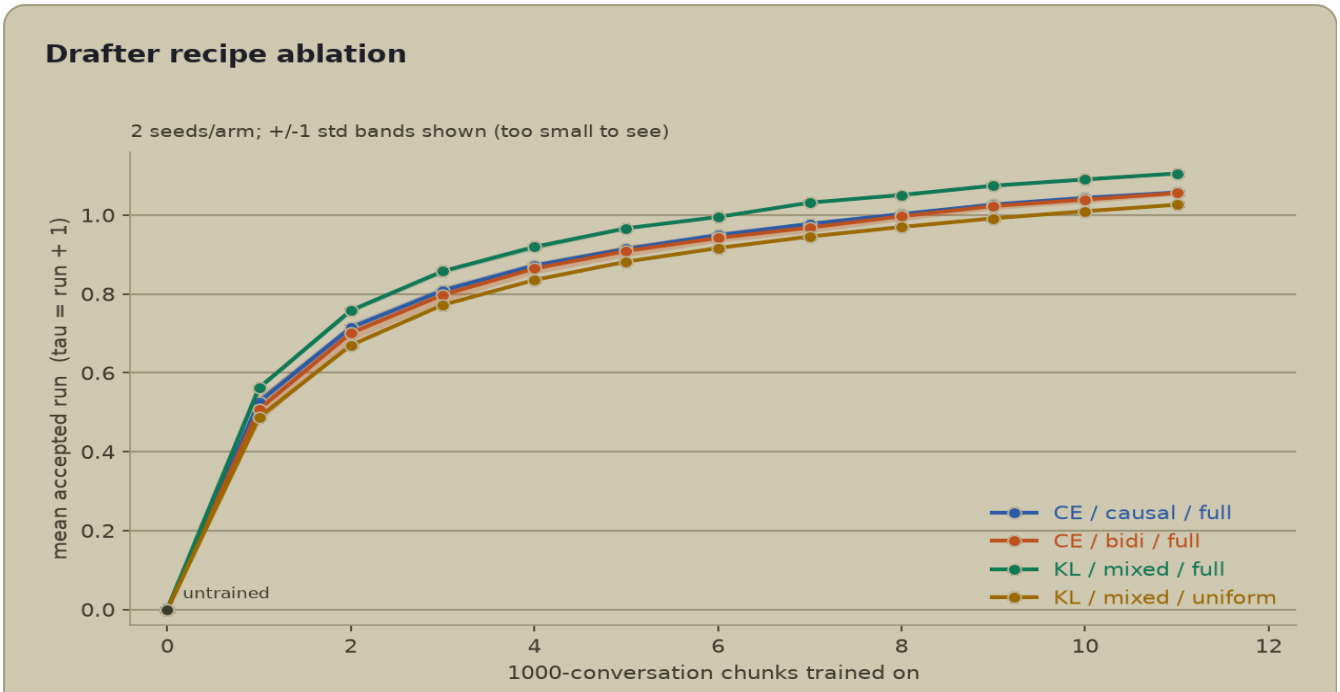
Three knobs stood out:

- **Loss.** The paper uses a position-weighted cross-entropy (Eq. 4: weight $w_k = \exp(-(k-1)/\lambda)$ at block position k, so early tokens count most), while Speculators (when I set this up in July 2026) defaulted to KL divergence, decayed the same way. KL divergence felt more intuitive since the drafter is, in a way, a distillation of the target, so I wanted to test how much this mattered. (I write λ for the paper's decay rate, which they call γ, to keep γ free for the block's draft count.)
- **Attention.** This one is harder to pin down. The paper has tokens attending *bidirectionally* within a block, and their PyTorch code[7] agreed at the time (`is_causal=False`; it has since been rewritten to match MLX). Their MLX code[7] masks the sliding layers causally, and Speculators defaults to causal too. On the released checkpoint's layer types, those last two give 5 causal layers plus 1 bidirectional. I tested all three. Causal seemed worth a shot since it's how the target factorizes, and acceptance is a prefix anyway.

- **Corruption** (the D in DFlash). Standard diffusion corrupts its input on a graded schedule, but DFlash masks the whole block at once, in training as much as when it drafts. I also tried the graded alternative, a uniform schedule: mask a random fraction of the block instead of all of it.

All three are free in the speedup formula (they raise τ without touching T_d or T_t), so they were the obvious place to start. I picked four promising configurations, trained one arm each, and watched acceptance as the training data piled up. Testing them meant a two-box homelab with a patch cable between them: one trains the drafter on a 16 GB card, the other runs the 35B-A3B target and serves its hidden states. (Getting to *train* the thing on such a small box was half the fun. Inference work usually feels a long way from hardware you can poke at.)

> Extracting the target's hidden states produces a lot of data: ~214 GB for 5K samples in bf16. Caching them in fp8 (e4m3) halves that. fp8 could plausibly corrupt the training signal, so I checked before committing: per-token cosine against bf16 is 0.9997. I only measured storage fidelity here; what fp8 does to acceptance is untested.



Mean accepted draft run (the accepted prefix, so $\tau - 1$) vs. training data, each recipe over 2 seeds. Arms are named loss_attention: the loss is KL or cross-entropy (ce), the attention is causal, bidirectional (bidi), or mixed (5 causal + 1 bidirectional layer). The trailing _full or _uniform is the mask: the whole block masked, which is DFlash's default, or that schedule.

KL with mixed causal/bidirectional attention (kl_mixed_full) won on both seeds. The two single-attention cross-entropy runs, causal and bidi, sat in the middle almost on top of each other, so attention doesn't seem to matter much for CE. Swapping kl_mixed_full's corruption to a uniform schedule (kl_mixed_uniform) dropped it below both of those CE runs. At inference the drafter only ever sees a fully-masked block, so maybe uniform corruption isn't acting as a regularizer here, just spending gradient on partially-masked inputs the drafter never meets.

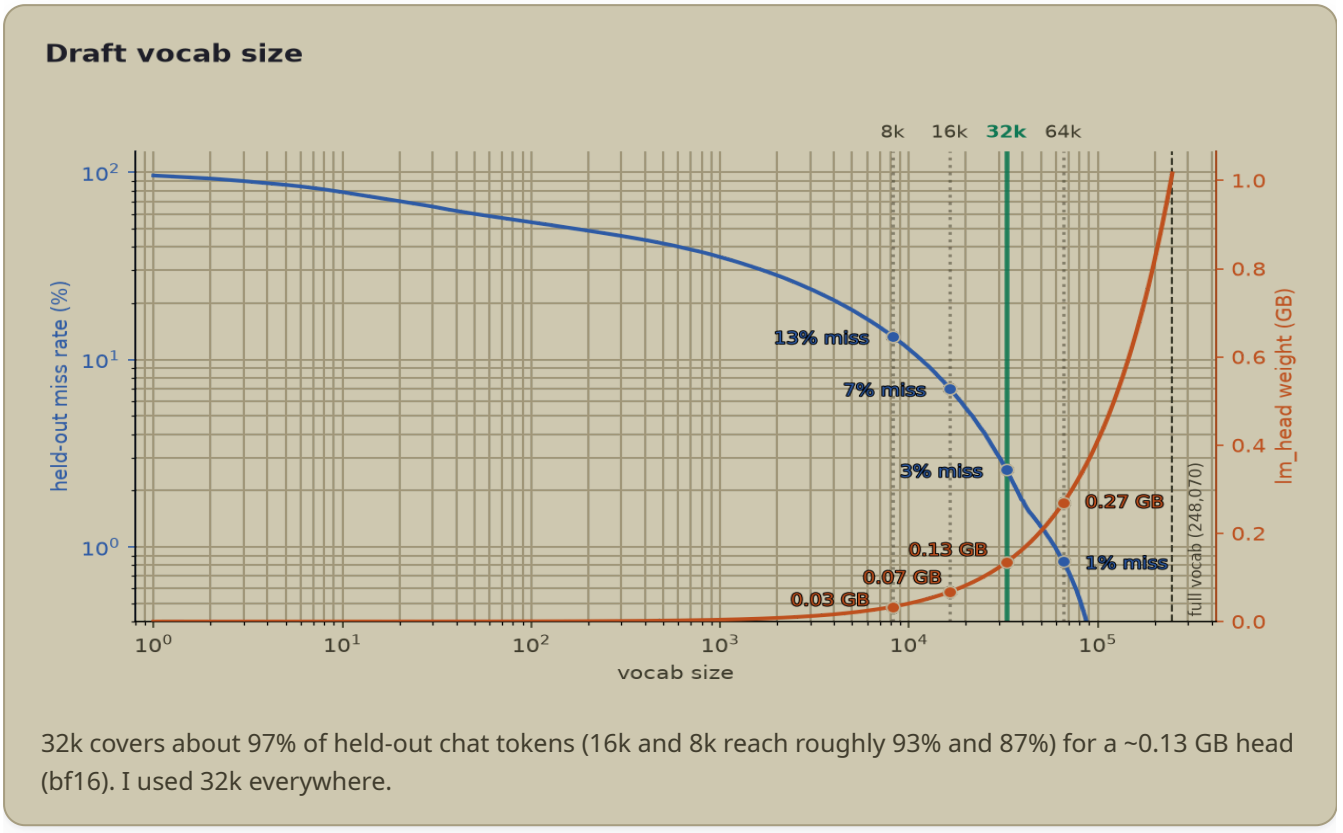
A fifth arm was meant to isolate the loss: ce_mixed_full, the same attention and mask as kl_mixed_full with cross-entropy instead of KL. It ran as a separate campaign that re-extracted its training features with a 1536-token conversation cap where the other four arms used 3072 tokens, so each of its chunks held about 57% of the re-

sponse tokens and 43% of the conversations were truncated. It scored worse than every other arm on the accept curve while matching ce_causal on the trainer's own accuracy, which is how the mismatch surfaced. It's off the plot until it's rerun. The only loss comparison left is kl_mixed_full against the single-attention CE arms, about 2% in τ, and that confounds loss with attention.

In a funny reversal, Speculators' default has since flipped. In August 2026 they switched the DFlash path to cross-entropy with a different position weighting, reporting that the new combination beat the old one[6]. Their change moved the loss, the weighting, the layer count and the block size together, so it doesn't isolate the loss either. My KL arm still led here, but upstream now disagrees.

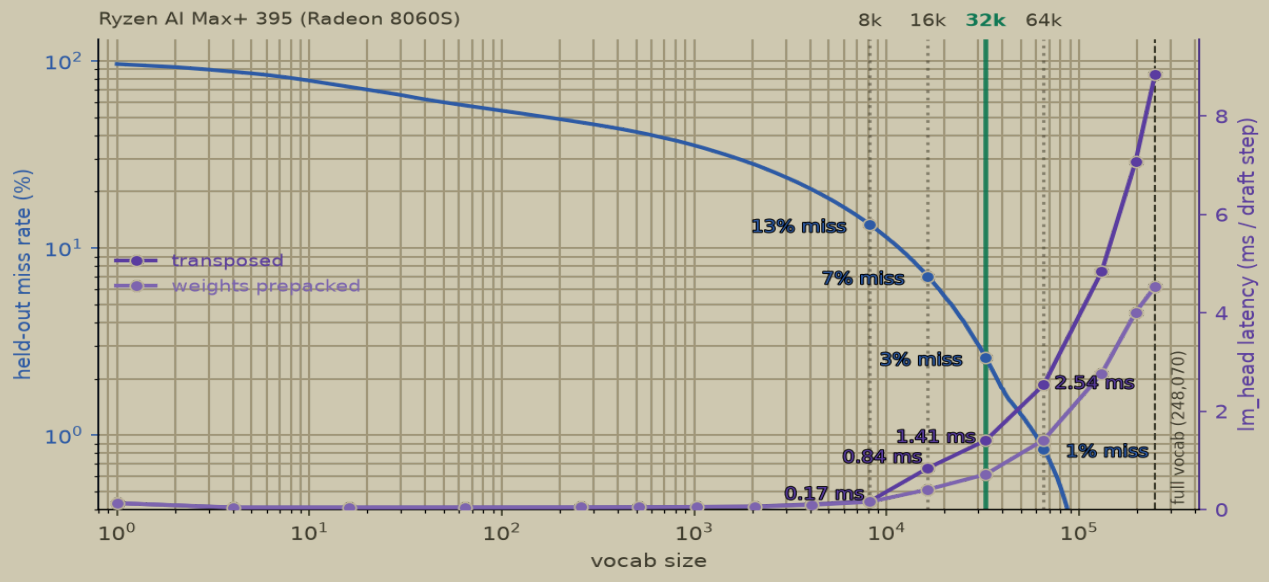
Should we shrink the vocab?

Speculators[6] lets you give the drafter a smaller vocabulary. Restrict it to the top-K most frequent tokens and its output head shrinks from `[2048×V]` to `[2048×K]`. That head gets streamed from memory on every draft step, so both its size and its latency fall roughly linearly in K: at the target's full 248k vocab it's 1.0 GB and 8.9 ms, at K = 32k it's 0.13 GB and 1.4 ms. And remember the drafter's quality doesn't touch the output sequence, only how fast you get there, so a smaller vocab can't cost quality. I picked the smallest K still covering ~97% of held-out chat tokens: K = 32k.



32k covers about 97% of held-out chat tokens (16k and 8k reach roughly 93% and 87%) for a ~0.13 GB head (bf16). I used 32k everywhere.

Draft vocab size



The same miss curve against draft-head latency. The head's [2048×V] weight is streamed every draft step, so latency climbs roughly linearly with V. Pre-packing that weight in a memory-friendly layout is ~2× faster in a microbenchmark at the full vocab.

This mattered less than I hoped, and it's not hard to see why. $T_d/T_t \approx 0.12$, so I was optimising something that was already cheap. Even driving $T_d \to 0$ leaves the verify at $T_t \approx 1.6 \cdot L_{target}$, which moves the ceiling from ~2.7× to ~3.0× and no further. Worth doing as a training optimisation, since it saves memory, but only if you're tight on VRAM. And that ~10% ceiling gain is probably optimistic to the point of being the wrong sign: 3% of tokens fall outside the 32k and are guaranteed rejections, which under a simple geometric model of acceptance costs about 10% of τ. Net, likely a loss.

Another complication with Qwen3.6-35B-A3B

Speculation is only lossless if you can actually undo a rejection, and on this target you can't do that for free. DFlash's decode loop crops the KV cache back to the accepted prefix every cycle. A full-attention layer crops cleanly. But Qwen3.6-35B-A3B also carries a recurrent state that advances through the whole block, and that can't be reverted — transformers still has no rollback for it[8].

So I patched the loop, in two steps that bracket the verify.

Where the writes were. In the stock loop a block verify pushes all 16 draft positions through the target in one forward. A KV layer appends 16 entries to its cache, and `crop()` trims them back to the accepted prefix afterwards. A linear-attention layer works differently. The cache holds one persistent copy of its `recurrent_states`, a single fixed-size tensor that has folded in every token so far. At the start of the forward the layer takes that tensor as the kernel's starting point; the kernel then carries its own running copy through all 16 positions and hands back the finished state; and at the end the layer calls `update_recurrent_state` once, which copies the finished state over the persistent one and drops the working copy (`update_conv_state` does

the same for the short conv buffer). One read and one copy-back per block, not per token. And unlike the KV cache, where rejected entries are just rows to slice off, there is no per-token record inside that tensor: nothing can be sliced off and there is no way to walk it back. Once the copy-back happens the rejected tokens are in it for good, and every later decode step starts from it.

Where the writes are now. The fix is to skip that copy-back during the verify. `_frozen_linear_states` does it by replacing `update_recurrent_state` and `update_conv_state` on each linear-attention cache layer with functions that copy nothing. The verify's forward is otherwise untouched: it reads the block-start state, computes the same outputs, and finishes with the cache still at block start. So after the verify the KV layers hold the whole block and the linear layers are where they were before it. `_advance_states` then brings both to the accepted prefix. For a KV layer that is `crop(keep_len)`. For a linear-attention layer it re-runs just `input_layernorm` and `linear_attn` over the accepted tokens, with the real `update_*` methods back in place, so this time the state is advanced and copied into the cache. The input to that replay is `hidden_states[i]`, the layer input the verify already computed, which is exact because position j only ever sees the positions before it, so the rejected tokens after it changed nothing. Every attention and MoE block is skipped, since their outputs *are* those saved hidden states.

```
@contextmanager
def _frozen_linear_states(cache):
    stashed = []
    for layer in cache.layers:
        if hasattr(layer, "recurrent_states"):        # linear-attn cache layer
            stashed.append((layer, layer.update_conv_state, layer.update_recurrent_sta
            layer.update_conv_state = lambda conv_states, *a, **k: conv_states       #
            layer.update_recurrent_state = lambda recurrent_states, *a, **k: recurrent_
    try:
        yield                                         # the block verify runs here
    finally:                                          # restore even if it raised
        for layer, conv_fn, rec_fn in stashed:
            layer.update_conv_state = conv_fn
            layer.update_recurrent_state = rec_fn

# after a frozen verify: kv layers hold the whole block, linear layers sit at block sta
def _advance_states(decoder_layers, cache, hidden_states, accepted: int, keep_len: int
    for i, cache_layer in enumerate(cache.layers):
        if hasattr(cache_layer, "recurrent_states"):
            layer = decoder_layers[i]
            # replay this layer over the accepted tokens; copy-back is live again
            normed = layer.input_layernorm(hidden_states[i][:, :accepted])
            layer.linear_attn(hidden_states=normed, cache_params=cache)
        else:
            cache_layer.crop(keep_len)                # kv layer
```

Snapshotting per-token states and restoring the accepted one would skip that recompute, but the chunked linear-attention kernel only hands back its final state. That route means patching the kernel, and I wanted to stay on the public interface.

The verify's own cost

That leaves T_t. A uniform speedup wouldn't move η at all, since it shrinks the baseline decode step by the same factor. But the block wakes ~47 experts against 8 for a single token, so anything that batches expert work cuts the verify and leaves the step alone.

And there was an easy one sitting there. The transformers implementation I was on runs the experts one at a time, in a Python loop inside the MoE block. Routing a handful of tokens to each of 47 experts is embarrassingly parallel, so that loop is pure waste. Each of those matmuls is tiny: a handful of routed tokens against a [2048×512] weight. In a warmed profile that forward was ~94% GEMM, with the expert loop's top kernel alone at 52.7%. (Recent transformers versions dispatch grouped matmuls by default instead.)

So I dropped in an off-the-shelf fused-MoE grouped-GEMM kernel[9] (tokens sorted by expert, then one grouped launch per projection instead of a loop), patched into the target's expert forward. It's token-identical on every prompt I've tried, about 3× on the expert matmuls and ~1.65× on DFlash decode overall.

Next steps

Everything above is preliminary:

- Two seeds, so the effect sizes are soft. The ordering held for both, but I can't say how big the gaps are.
- $\tau \approx 4.9$ and 2.1× come from the drafter z-lab released; my own arms are not fully trained so they're currently lower.
- A rerun of the bench on later code gave $\tau = 5.44$ instead of 4.92. Running today's code twice back to back reproduces 5.44 exactly, so the gap tracks the script change between the runs, not run-to-run noise; the July version was never committed, so I can't split it further. I keep 4.9 because it is the run that produced the 17.4 tok/s.
- I timed the draft head on its own rather than measuring the vocabulary choice end to end, so the speedup was a projection. It also ignores that tokens outside the 32k are guaranteed rejections, which caps τ; I never measured that cost, so the net effect could be negative.

My plan for the next round was to spend losslessness: let the draft run long and patch its mistakes with confidence-guided refinement. Reading around killed both halves of that. Relaxing the verification isn't worth it: an evaluation of lossy verification[10] found that relaxation silently rewrites the decoding distribution and can badly degrade quality. The one rule they build that matches lossless quality on their benchmarks gains 3.7% in accepted tokens per block. And the patching never needed the relaxation: D²SD[11], which builds on DFlash, already uses drafter confidence to decide where to re-draft.

I want to condition the block up front on a token predicted to land somewhere ahead of the current position. I don't know yet whether that signal is there.

Part 2: measuring the fast engine (August 2026)

Part 2 has its own page: the same speculation measured on llama.cpp/Vulkan, where a verify costs ~4.2× a decode step instead of 1.6×, plus the two upstream bugs that checking losslessness turned up.

References

1. [a b] Qwen. Qwen3.6-35B-A3B model card: 40 layers, hidden layout 10 × (3 × (Gated DeltaNet → MoE) → 1 × (Gated Attention → MoE)), 256 experts, 8 routed plus 1 shared activated.
2. Yaniv Leviathan, Matan Kalman, Yossi Matias. *Fast Inference from Transformers via Speculative Decoding.* arXiv:2211.17192, 2022. Relied on for losslessness: "we can make exact decoding from the large models faster … without changing the distribution."
3. [a b c] Jian Chen et al. *DFlash: Block Diffusion for Flash Speculative Decoding.* arXiv:2602.06036, February 2026. Block attention: "Tokens attend bidirectionally within the same block and to the corresponding injected target context features, while attention across different blocks is disallowed." Headline: 6.08× on Qwen3-8B, MATH-500, greedy. Table 3, Qwen3-Coder-30B-A3B at batch size 1: 3.5× at τ 8.09 (HumanEval), 2.6× at τ 6.42 (LiveCodeBench), 3.2× at τ 7.23 (MBPP).
4. z-lab. Qwen3.6-35B-A3B-DFlash, config.json: `num_hidden_layers` 6, `hidden_size` 2048, `target_layer_ids` [1, 6, 11, 16, 22, 27, 32, 37] of `num_target_layers` 40, `layer_types` five `sliding_attention` then one `full_attention`, `block_size` 16.
5. ggml-org/llama.cpp. PR #22105, *[Speculative decoding] feat: add DFlash support.* Merged 2026-06-28.
6. [a b c d] vLLM project. Speculators. Draft vocabulary: `src/speculators/train/vocab_mapping.py`. Training defaults changed in PR #980, merged 2026-08-13, per RFC #979: `--loss-fn` kl_div to ce, `--per-position-loss-weight` fixed-exp-decay to dpace, `--num-layers` 1 to 5, `--block-size` 8 to 16, described as "the settings that consistently helped across every DFlash configuration we tested." The RFC recommends "Causal sliding-window attention (SWA) … applied to all draft layers."
7. [a b c] z-lab. dflash/model.py at 7d2ea1e9: the block forward passes `is_causal=False`. The current file instead sets `is_causal = layer_type == 'sliding_attention'` per layer. dflash/model_mlx.py: `is_causal = is_sliding` unless the config overrides it, and a full-attention layer with `is_causal` false gets no mask.
8. Hugging Face transformers. PR #45846, *Add Cache.snapshot() / Cache.restore(snapshot) for tentative-forward rollback.* Closed without merging; its description: "there has been no supported way to undo a forward pass."
9. woct0rdho. transformers-qwen3-moe-fused. "The Qwen3 MoE model (and all other MoE models) in HF Transformers is notoriously slow, because it uses a for loop to access the experts." The replacement is a Triton grouped GEMM: "The implementation in this repo is largely based on the Triton grouped GEMM," with tokens sorted by expert first.
10. Tianyu Wang et al. *Revisiting Lossy Verification in Speculative Decoding: Mechanisms, Trade-offs, and Failure Modes.* arXiv:2607.26627, July 2026. "Such relaxation silently rewrites the decoding distribution, and the resulting acceleration can come at the cost of unstable, sometimes severely degraded generation quality."

11. Liyuan Zhang et al. *D^2SD: Accelerating Speculative Decoding with Dual Diffusion Draft Models.* arXiv:2606.04446, June 2026. On DFlash: "We adopt it as our first-stage drafter." On confidence: "the first diffusion drafter generates a block along with per-position confidence scores that are used to identify the most likely rejection boundary."