Modern AI Systems: Agents and System Optimizations

LLM Serving: Speculative Decoding

Juncheng Yang



Harvard John A. Paulson
School of Engineering
and Applied Sciences



HARVARD UNIVERSITY
MaDSys
Measurement and Design of Systems Lab

Logistics

- Course Overview
  - Last serving lecture (Nov 4): batching, paging and parallelism
  - Today: speculative decoding; next: KV cache, then quantization
- Logistics
  - Assignment 5 (Mads-Stack) is out today; due Tue, Nov 24, 11:59 pm
  - Wed, Nov 18: showcase your Assignment 4 serving engine
  - No class Wed, Nov 25 (Thanksgiving recess)

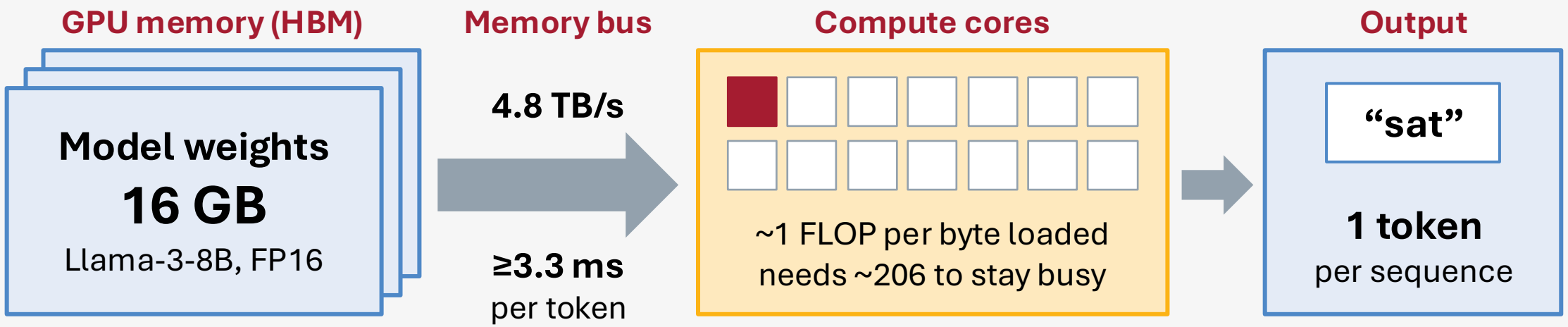
Plan for today

- Foundation of speculative decoding
- Drafter
- Verification
- Systems

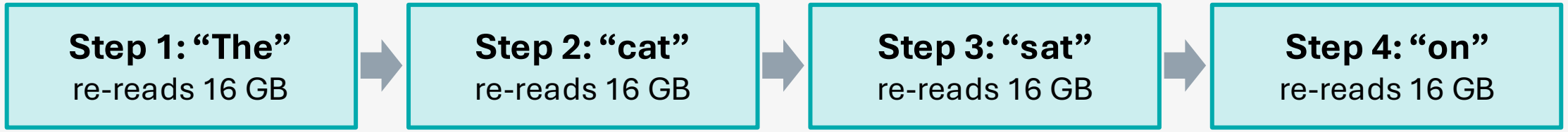
Foundations

- why decoding is slow
- draft, verify, accept
- rejection sampling and exactness
- the arithmetic of speedup

Decoding is memory-bound



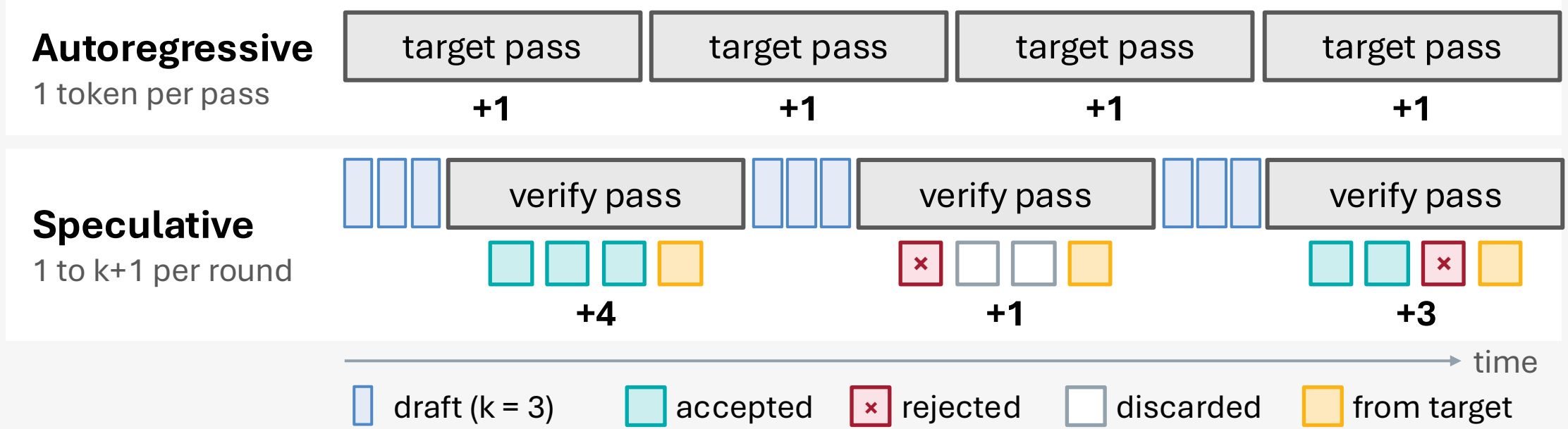
Tokens come out one at a time: each step waits for the one before



Question: The cores sit idle while weights stream in. What could use that compute?

Draft cheaply, verify in parallel

1 Draft: a small model guesses k tokens → **2 Verify:** the big model checks all k at once → **3 Accept:** accept the right guesses, add one token



In the same time, speculative decoding makes 8 tokens (4 + 1 + 3) instead of 4.

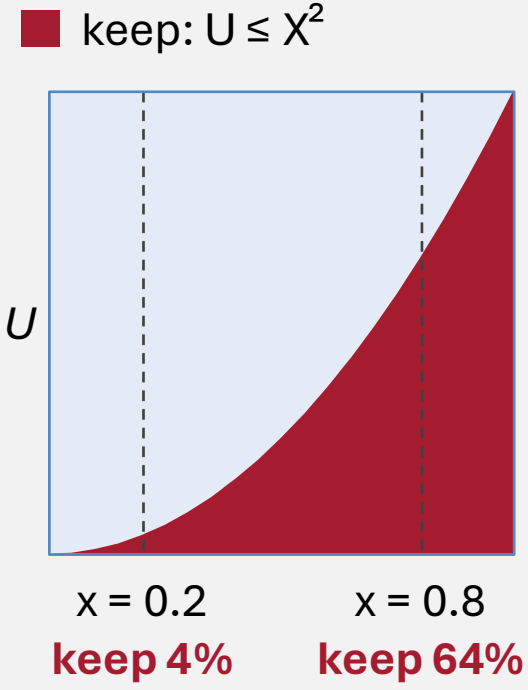
Question: The drafter is a weaker model. How can the output still match the target?

Fast Inference from Transformers via Speculative Decoding
Accelerating Large Language Model Decoding with Speculative Sampling

Rejection sampling makes cheap guesses exact

Example: $p(x) = 3x^2$ on [0, 1]

Draw X, U uniform on [0, 1]; keep X if $U \le X^2$.



Kept X has density $\propto x^2$.

Over [0, 1] the integral of x^2 is ⅓, so normalizing gives $3x^2$.

Red is a third of the square: 1 in 3 kept.

General rule

Pick a proposal q you can sample and a constant M with **$p(x) \le M \cdot q(x)$** for all x.

Draw $X \sim q$; keep it with chance **$p(X) / (M \cdot q(X))$**. Otherwise, retry.

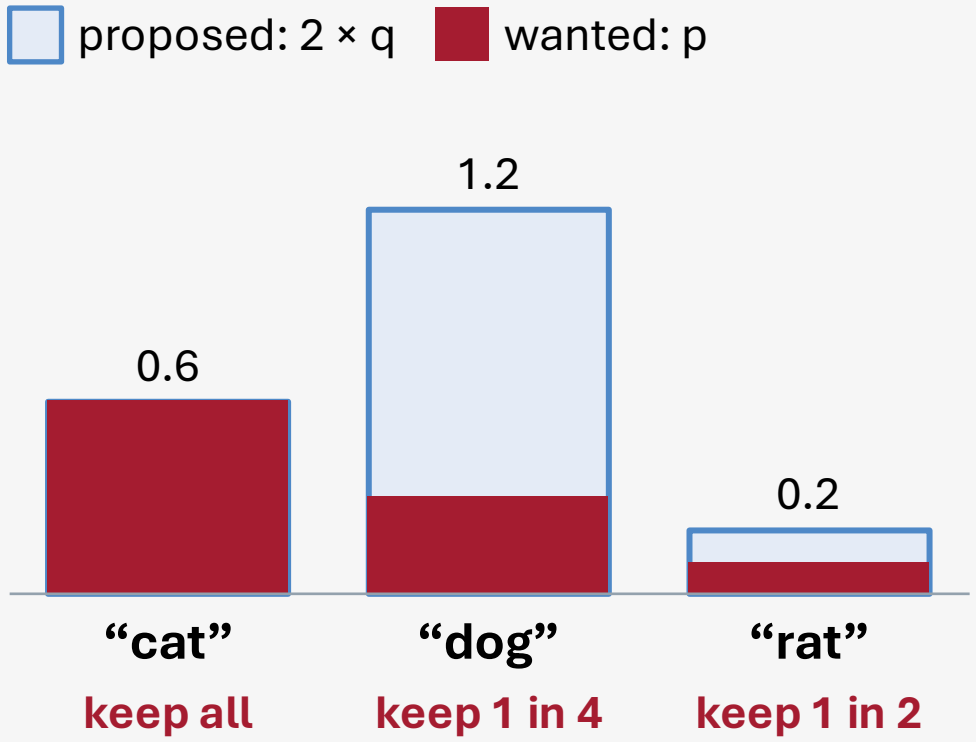
Why it works

$$q(x) \times \frac{p(x)}{Mq(x)} = \frac{p(x)}{M}$$

How often x is proposed × chance it is kept leaves p(x)/M, so kept samples follow p. 1 in M draws is kept: a q close to p wastes fewer.

Rejection sampling works on tokens

With M = 2, blue covers red



Throw away the pale part; red is left.

1 Propose: the draft model q picks a word.

2 Check: keep it with chance red ÷ blue: p(x) / (2·q(x)).

3 Retry: if it is thrown away, pick again.

Same rule, M = 2: kept words follow p, at 2 picks per word on average.

Speculative decoding’s version

It drops the ×2: keep a draft token with chance min(1, p/q). No retry: on a rejection, sample from p, so every round makes a token.

Various Techniques Used in Connection with Random Digits
Fast Inference from Transformers via Speculative Decoding

Each round drafts, verifies and accepts

1 Draft: the small drafter q guesses k tokens $\tilde{x}_1 \ldots \tilde{x}_k$, one at a time. The tilde marks a guess.

2 Verify: one pass of the big target p scores all k guesses, plus one position after them.

3 Accept: accept each $\tilde{x}_i$ with chance (1), left to right. At the first rejection, resample from (2) and stop.

4 Bonus: if all k guesses survive, take one more token from the target's last pass. It is free.

(1) Accept a guess

$$\text{accept } \tilde{x}_i \text{ with probability } \min\left(1, \frac{p(\tilde{x}_i)}{q(\tilde{x}_i)}\right)$$

p(x) and q(x) are each model's chance of token x. If p ≥ q, always accept.

(2) Resample after a rejection

$$x \sim p'(x) = \frac{\max\left(0, p(x) - q(x)\right)}{\sum_{x'} \max\left(0, p(x') - q(x')\right)}$$

Tokens p wants more than q offered, rescaled to sum to 1.

Fast Inference from Transformers via Speculative Decoding
Accelerating Large Language Model Decoding with Speculative Sampling

Accept with probability min(1, p/q)

$$\text{accept chance} = \min\left(1, \frac{p(\tilde{x})}{q(\tilde{x})}\right)$$

If the target likes the guess at least as much as the drafter, always accept.

Example: a vocabulary of three tokens; the drafter guesses one of them.

token	target p	drafter q	accept chance	why
“cat”	0.6	0.3	1	q < p: always accept
“dog”	0.3	0.6	½	q > p: accept 1 in 2
“rat”	0.1	0.1	1	q = p: always accept

Takeaway: only over-guessed tokens (“dog”) get rejected, and only by the amount q exceeds p.

Rejected tokens are resampled from the leftover

$$p'(x) = \frac{\max\left(0, p(x) - q(x)\right)}{\sum_{x'} \max\left(0, p(x') - q(x')\right)}$$

After a rejection, draw only from tokens the target wanted more than q offered. Same cat / dog / rat example.

1 Leftover: max(0, p − q)

cat	0.3
dog	0
rat	0

total leftover = 0.3

2 Rescale to sum to 1

cat	1.0
dog	0
rat	0

0.3 ÷ 0.3: all on “cat”

3 Why it fits

Accept rate α = Σ min(p, q) = 0.3 + 0.3 + 0.1 = 0.7

Rejections happen 1 − α = 0.3 of the time, the same 0.3 as the leftover total.

So resampling adds exactly the leftover: +0.3 to “cat”.

Takeaway: the resample refills exactly the probability that rule (1) rejected.

Accepted plus resampled equals p

$$\Pr[\text{emit } x] = \min\big(p(x), q(x)\big) + \max\big(0, p(x) - q(x)\big) = p(x)$$

Chance each token comes out, by path



Check each token

cat: 0.3 + 0.3 = 0.6 = p ✓

dog: 0.3 + 0 = 0.3 = p ✓

rat: 0.1 + 0 = 0.1 = p ✓

Same totals as the target p.

All teal = α: 0.3 + 0.3 + 0.1 = 0.7

Takeaway: for every token the two paths add back to p, so the output follows p exactly.

Speculative decoding is exact

A token x can come out two ways: **guessed and accepted**, or **rejected and resampled as x**.

$$\Pr[\text{emit } x] = \min\big(p(x), q(x)\big) + \max\big(0, p(x) - q(x)\big) = p(x)$$

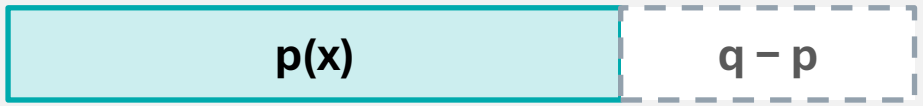
chance x comes out | guess x accepted | rejected, then resampled as x | target p's chance

When q(x) < p(x): drafter too cautious



All of q(x) is accepted; resampling adds p − q.

When q(x) > p(x): drafter too eager



Only p(x) is accepted; the extra q − p is rejected.

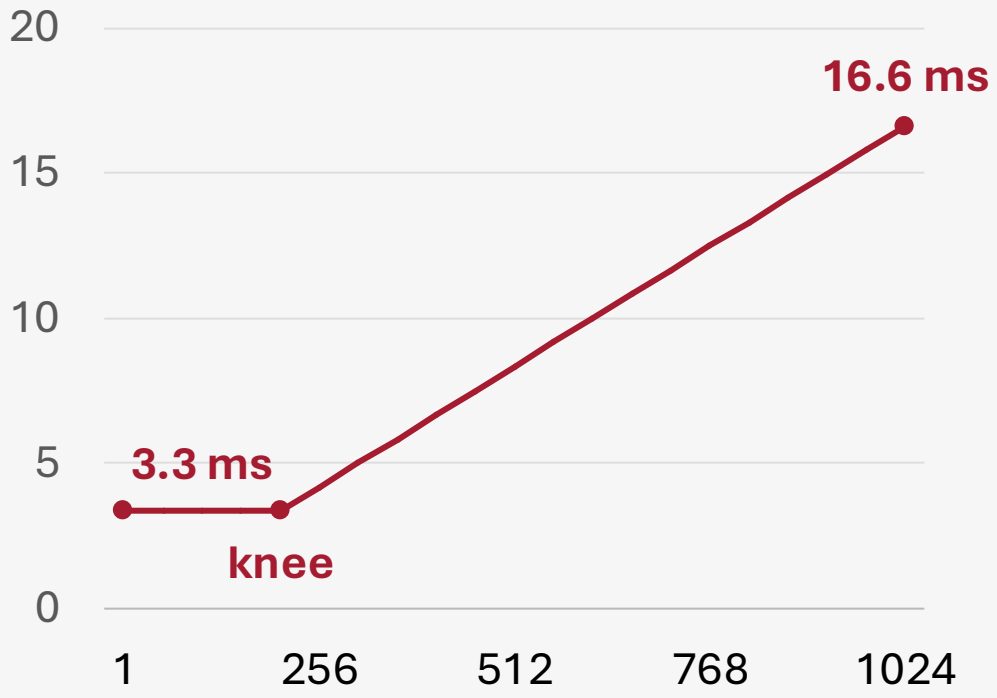
Question: The output is exact. But what does checking k extra tokens cost?

Fast Inference from Transformers via Speculative Decoding
Accelerating Large Language Model Decoding with Speculative Sampling

Verification cost is low

Time per pass, ms, by tokens in the pass

Llama-3-8B FP16 on one H200, ideal roofline



$$T(n) \approx \max(W/B,\ n \cdot W/F)$$

- n tokens per pass
- W weights, 16 GB
- k drafts per request
- B bandwidth, 4.8 TB/s
- F compute, 989 TFLOPS
- $batch$ requests per pass

The knee is n = F / B ≈ 206

Each weight costs 2 bytes and 2 FLOPs per token, so n tokens cost n·W FLOPs.

Batch and k share the budget

n = batch × (k + 1) ≤ 206 stays free: k ≤ 5 at batch 32, k ≤ 2 at 64, none from 128.

Acceptance rate sets the speedup

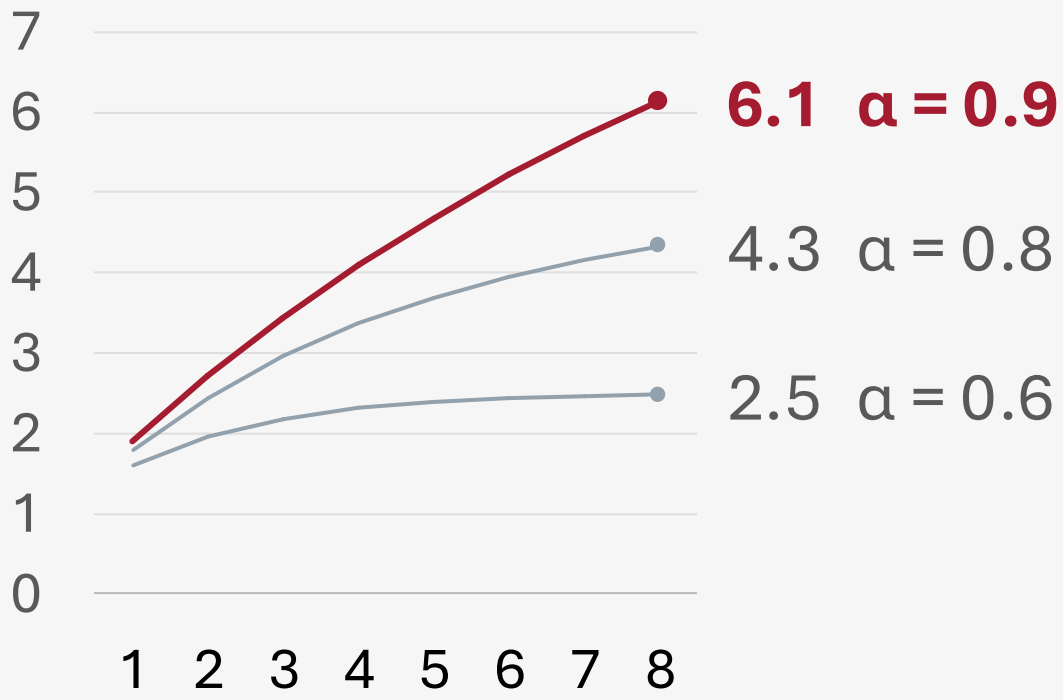
α chance a draft token is accepted
k draft length per round
c draft step cost ÷ target step cost

Expected tokens per target pass
$E = 1 + \alpha + \ldots + \alpha^k = (1 - \alpha^{k+1}) / (1 - \alpha)$

Speedup over plain decoding
$S = E / (ck + 1)$

Example: $\alpha = 0.8$, $k = 4$, $c = 0.05$ gives $E \approx 3.4$ and $S \approx 2.8\times$

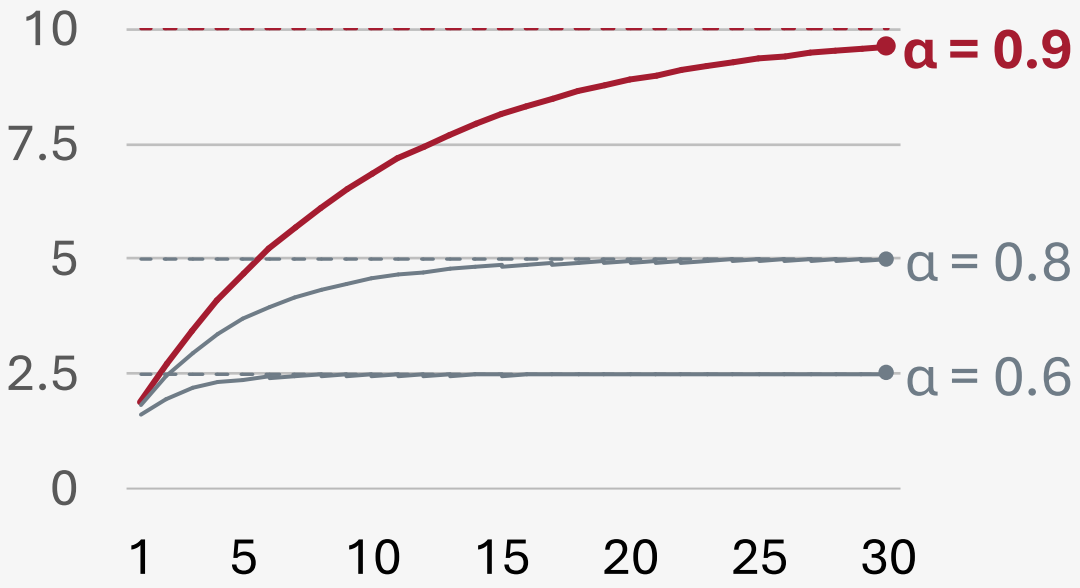
Expected tokens per target pass
by draft length k, for three acceptance rates α



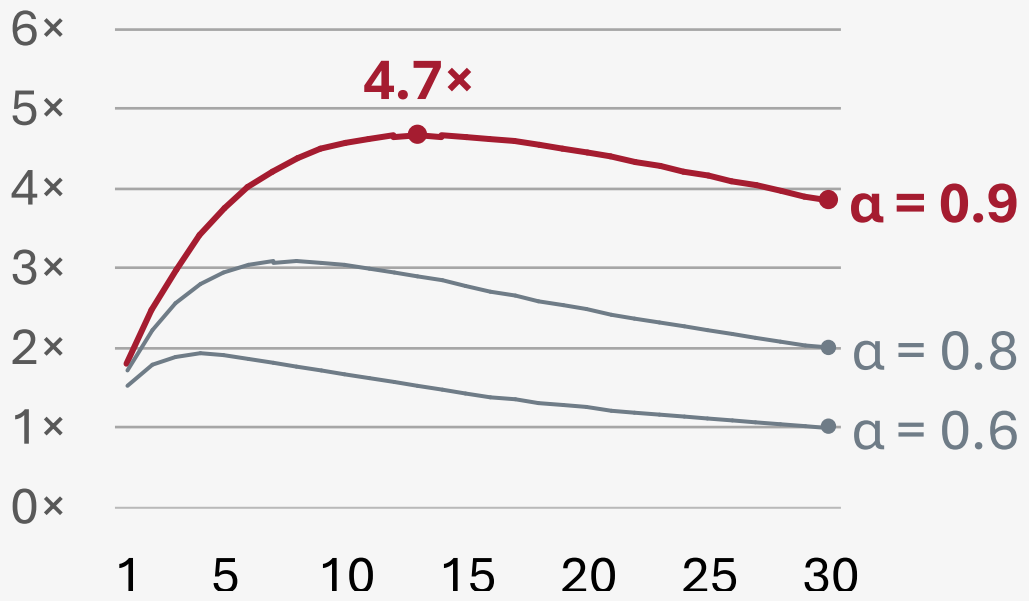
Drafting further has diminishing returns

As k → ∞: $\alpha^{k+1} \to 0$, so $E \to 1 / (1 - \alpha)$.
But: each draft step still costs c, so $S = E / (ck + 1)$ peaks, then falls toward 0.

Expected tokens per target pass, E
by draft length k, for three acceptance rates α



Speedup over plain decoding, S
by draft length k, draft cost c = 0.05

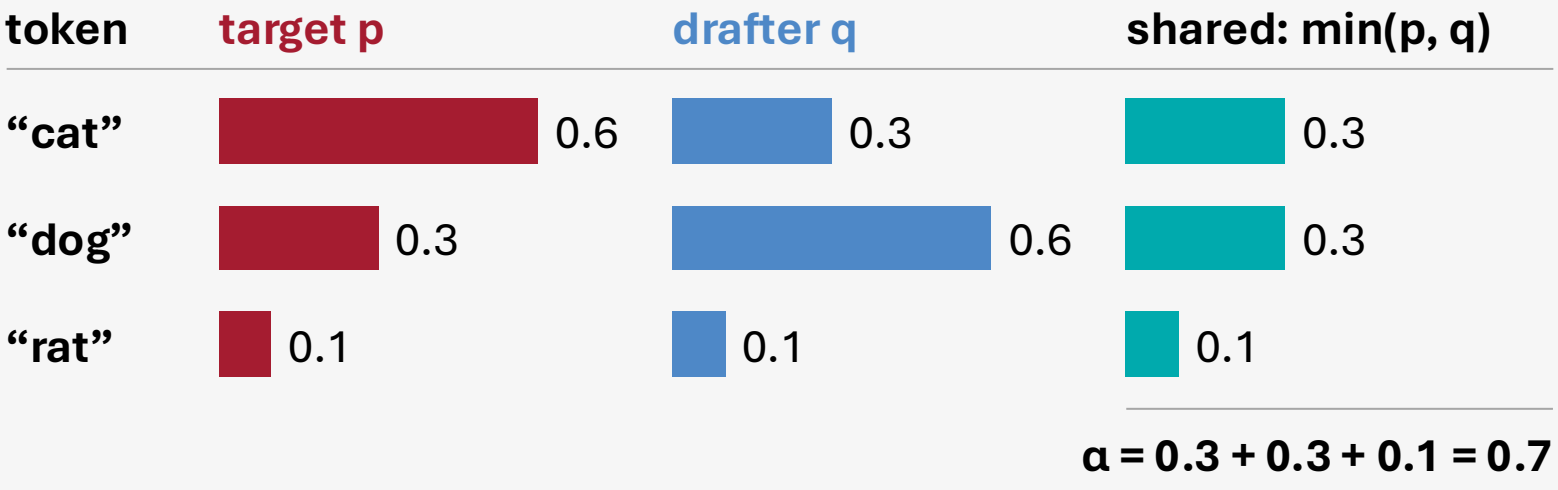


α measures how much p and q overlap

$$\alpha = \sum_{x} \min\big(p(x), q(x)\big)$$

Each token counts up to the smaller of its two chances: α is the probability both models share.

Example: the same cat / dog / rat vocabulary.

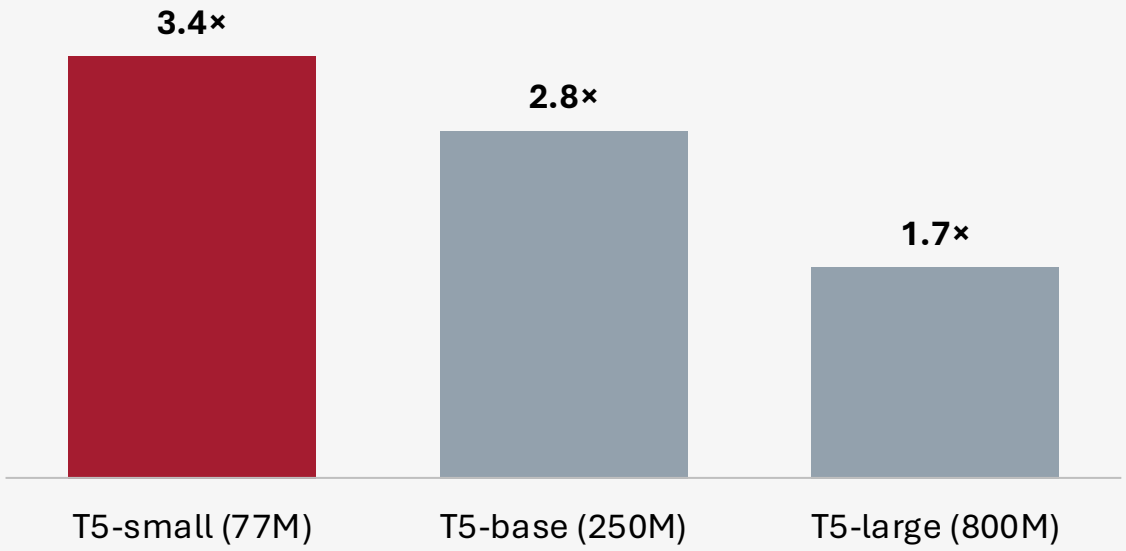


Greedy decoding

At temperature 0 each model puts all its chance on one token, so α is how often the drafter’s top token matches the target’s.

Smaller drafters gave bigger speedups

Speedup over T5-XXL (11B) decoding, by drafter
English–German translation, greedy decoding



Why the bigger drafters lost

α barely moves: 0.75, 0.80, 0.82.

Draft cost grows 10×: 77M to 800M parameters, so c·k eats the gain.

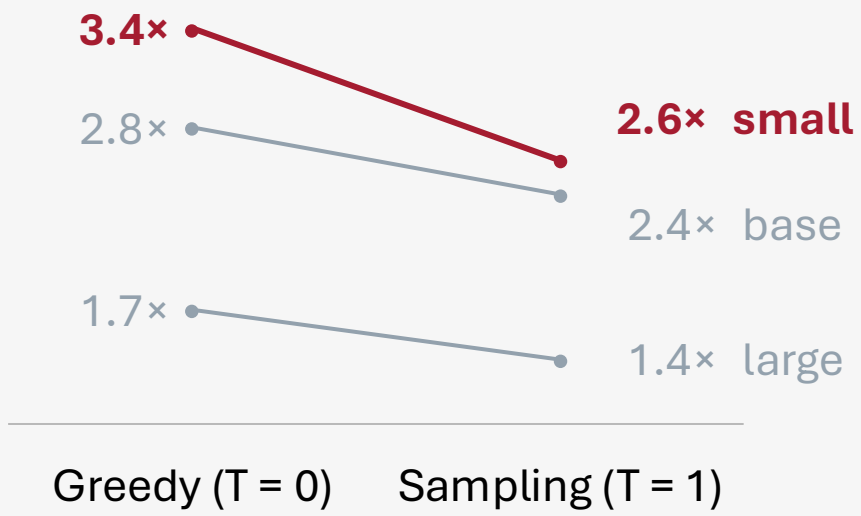
Takeaway: a cheap drafter with decent α beats an accurate but slow one.

Fast Inference from Transformers via Speculative Decoding
Accelerating Large Language Model Decoding with Speculative Sampling

Higher temperature lowers the speedup

Speedup over T5-XXL (11B) decoding, by drafter

English–German; α fell 0.75→0.62, 0.80→0.68, 0.82→0.71



T = 0 checks one token

Accept iff the draft is argmax p, so α is the top-1 match rate.

T = 1 checks the whole shape

Any mismatch in the distribution reduces acceptance.

The best k shrinks too

At T = 1 the best k fell from 7 to 5.

Three levers raise the speedup

$$S = \frac{E}{ck + 1}$$

E tokens per target pass, set by α and k
c·k cost of the k draft steps, in target steps
1 the verify pass, assumed to cost one decode step

Raise E

Drafters that match p better raise α. Token trees try several guesses per position. Relaxed checks accept near-misses.

Shrink c·k

Reuse the target's hidden states, copy text already seen, or draft all k tokens in one pass.

Keep the 1 at 1

Checking k + 1 tokens costs one decode step only while compute sits idle. Large batches and MoE models break this.

Draft models

Most drafters reuse the target model

Drafts come from	Family	Examples
A separate model	**Independent small model**	T5-small, Llama 3.2 1B
The target's own layers or hidden states	**Self-speculation**	LayerSkip, SWIFT
	Prediction heads	Medusa
	Feature-level drafters	EAGLE, EAGLE-2, EAGLE-3
	Native multi-token prediction	DeepSeek-V3 MTP
	Parallel and block drafters	DFlash, DSpark
Text already seen	**Copying and retrieval**	N-gram lookup, SuffixDecoding

DistillSpec trains the drafter to agree

Problem Off-the-shelf small models are trained for quality, not for agreement with a given target, so their acceptance rate α is low and caps the speedup.

Motivation α measures how much probability the drafter shares with the target, so a drafter distilled to match the target gets a higher α.

Key idea Distill from the target on prefixes the drafter itself samples (on-policy), with a divergence chosen per task and decoding strategy.

Question: Could the target model draft for itself, with no second model?

LayerSkip drafts with early layers

Problem

Speculative decoding usually needs a second model: extra weights and KV cache, a matching tokenizer, and a drafter to retrain whenever the target changes.

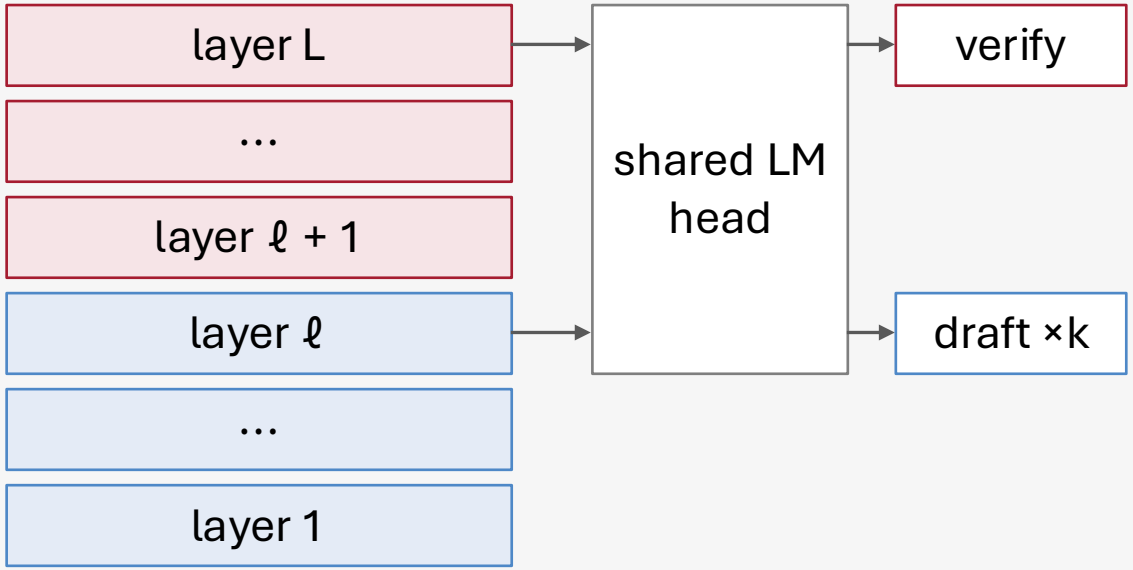
Motivation

Many tokens don’t need every layer, but a normally trained model’s middle layers can’t decode through the LM head, so exiting early gives poor drafts.

Key idea

Train with layer dropout and an early-exit loss so early layers can predict; draft with layers 1 to ℓ, verify with layers $\ell + 1$ to L, reusing the drafts’ KV cache.

LayerSkip verifies only the skipped layers



cost ≈ k·ℓ/L + (1 − ℓ/L) passes
vs k·c + 1 with a separate drafter

Insights

One model, one cache: no drafter weights or second KV cache; verification reuses the KV of layers 1..ℓ.

Training makes exits usable: layer dropout that rises with depth, plus an exit loss on the shared LM head.

Training-free: SWIFT skips middle layers instead, chosen by Bayesian optimization.

LayerSkip: Enabling Early Exit Inference and Self-Speculative Decoding
SWIFT: On-the-Fly Self-Speculative Decoding for LLM Inference Acceleration

Medusa adds heads that guess ahead

Problem

A separate draft model is hard to get and to serve: it must share the tokenizer, track the target's distribution, and run beside it as a second model.

Motivation

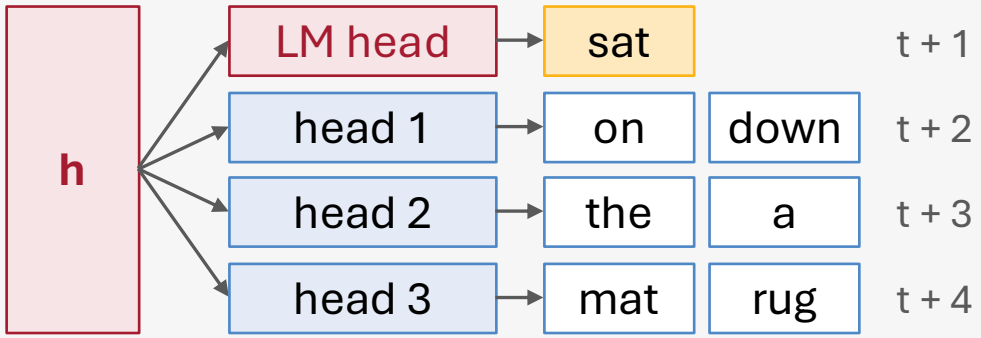
The last hidden state already carries signal about tokens beyond the next one, and small heads on it train cheaply, even with the backbone frozen.

Key idea

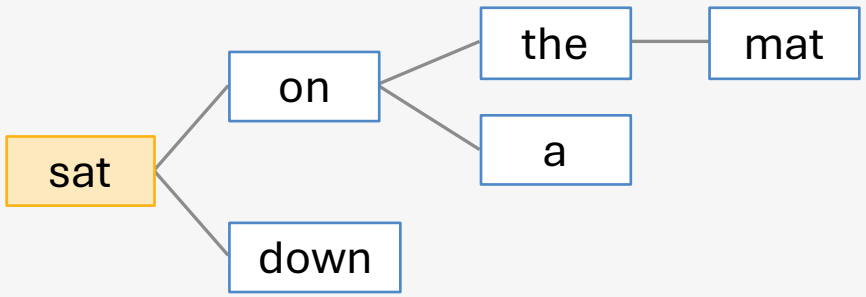
Add K heads on the last hidden state; head k predicts token t + k + 1. Combine each head's top guesses into a tree and verify it in one pass with tree attention.

Medusa heads lose accuracy with depth

One target pass: every head reads the same h



Candidate tree from the heads' top guesses



Kept: the paths most often accepted on calibration data

Insights

Blind guesses: every head reads the same h, so head 3 can't see heads 1–2's picks; accuracy falls with depth.

Trees recover some accuracy: top guesses per head, pruned to the most-accepted nodes, verified in one pass.

Two training modes: frozen backbone (Medusa-1) or joint fine-tuning (Medusa-2).

Question: Later heads cannot see earlier guesses. What if each guess could?

EAGLE drafts from the target's features

Problem

Token-level drafters accept few tokens: small models predict the next token poorly, and Medusa's heads can't see each other's guesses.

Motivation

Second-to-top-layer features are more regular than tokens, so they extrapolate better, but after sampling the next feature is ambiguous unless the drafter knows the chosen token.

Key idea

Predict the target's next feature, not the next token, and give the drafter the sampled token so it knows which way the text went.

EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty
EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees
EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test

Each EAGLE version closed one gap

EAGLE-1
ICML 2024

Gap: small drafters guess tokens poorly.

Fix: draft features, feed back the sampled token.

EAGLE-2
EMNLP 2024

Gap: one fixed tree for every context.

Fix: grow the tree by drafter confidence.

EAGLE-3
NeurIPS 2025

Gap: training never saw its own guesses.

Fix: train on its own outputs; fuse three layers.

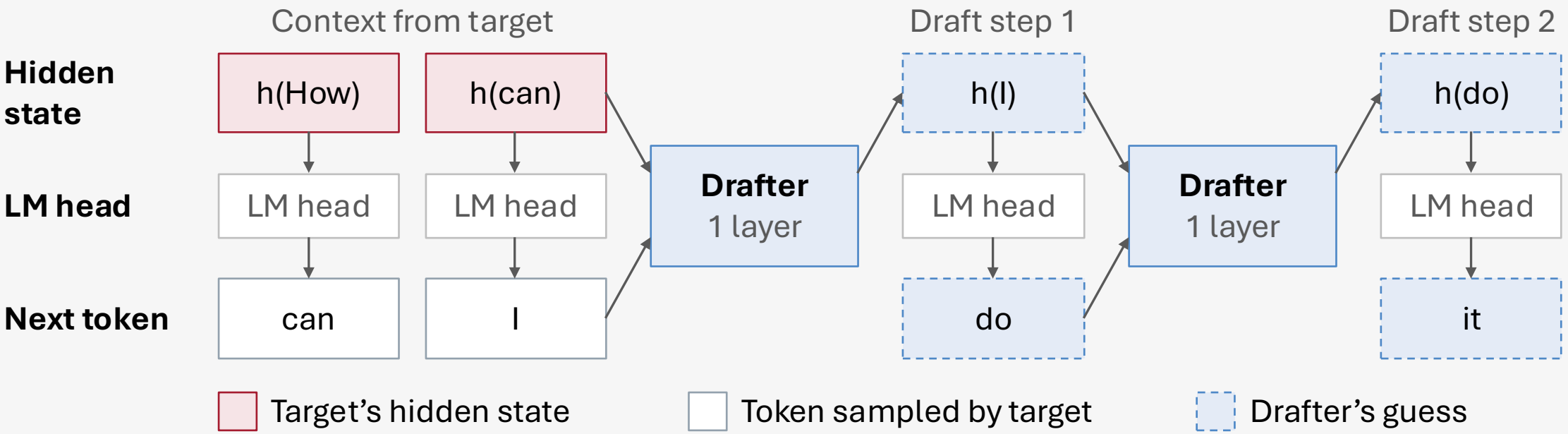
EAGLE-4
Preprint 2026

Gap: one tree size for every load.

Fix: size each tree from measured cost.

Li et al.: EAGLE (ICML 2024), EAGLE-2 (EMNLP 2024), EAGLE-3 (NeurIPS 2025); EAGLE-4 Team (2026)

EAGLE-1 drafter: one layer, fed its own guesses



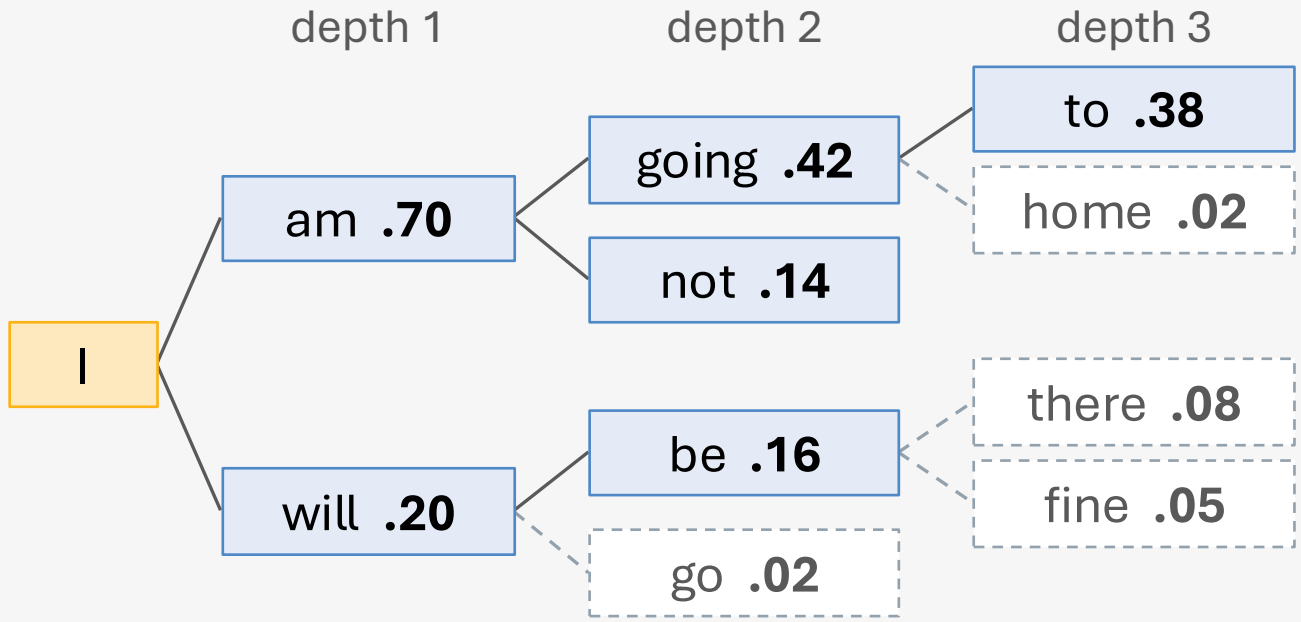
Pair: each input joins a hidden state with the token that came out of it.

Draft: one decoder layer predicts the next hidden state; the target's frozen LM head turns it into a token.

Repeat: after the first guess, the drafter feeds back its own predicted hidden states.

EAGLE-2 grows a confidence-shaped tree

Node value = product of drafter confidences on its path



Insights

Confidence ≈ acceptance: the drafter is calibrated.

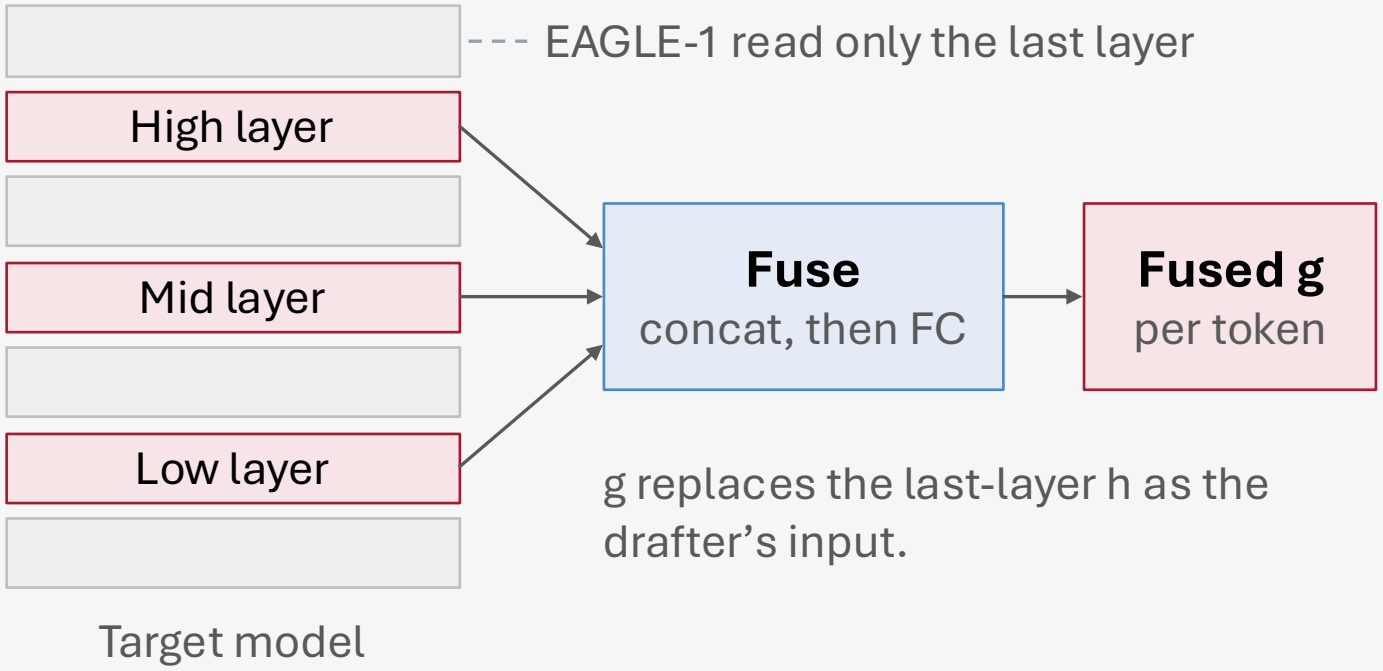
Expand, then rerank: grow the top-k nodes at each depth, then verify the m nodes with the highest V (here k = 2, m = 6; the paper uses k = 10, m = 48–60).

Same drafter: no retraining; 20–40% faster than EAGLE-1.

Context sets the shape: the same budget goes deep on predictable text and wide where the drafter is unsure.

EAGLE-3 fuses low, mid and high layers

The drafter reads three layers of the target



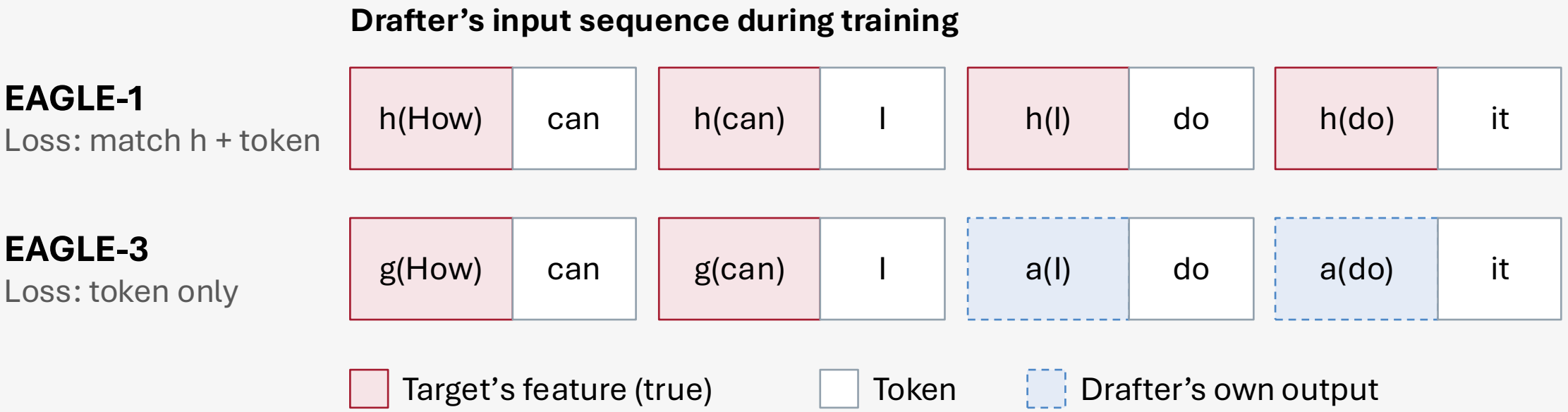
What EAGLE-3 changes

Predict tokens, not features: the feature loss is gone.

Fuse three layers: low, mid and high features carry more than the top layer.

Train as you infer: later steps learn from the drafter's own outputs.

EAGLE-3 trains on its own drafted outputs



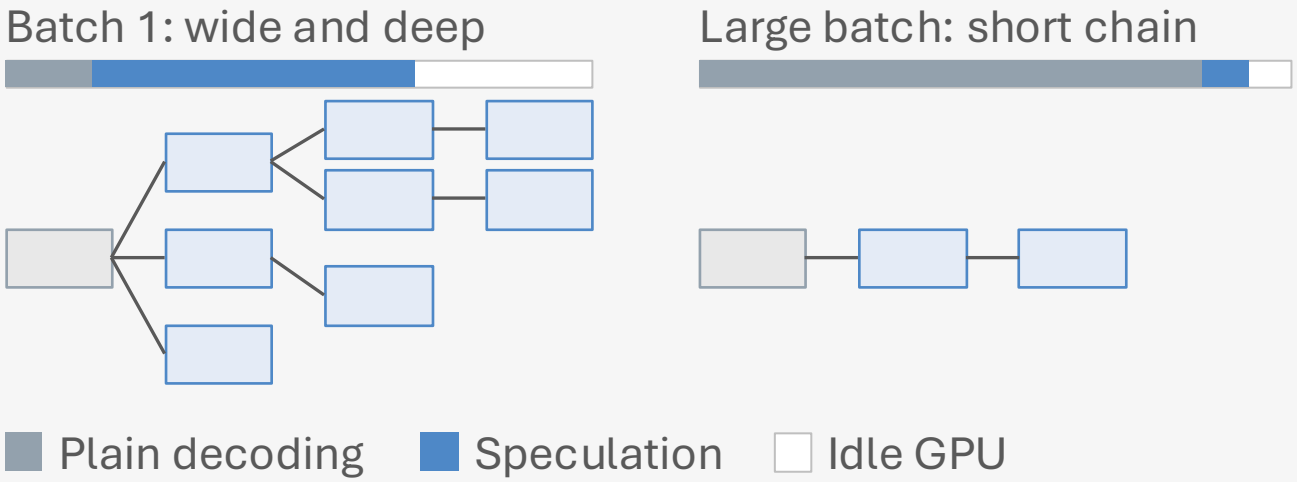
EAGLE-1: trained only on true hidden states, so later draft steps never practised on their own guesses.

EAGLE-3: from the third position on, it reads its own outputs a(·), exactly what it sees at inference.

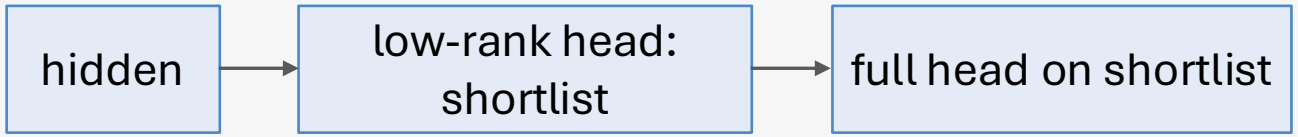
Token loss only: with no hidden state to match, the drafter is free to read the fused g.

EAGLE-4 sizes each tree from measured cost

Same drafter, tree sized to the load



Cheaper LM head for the drafter



Draft cost down over 40%, full vocabulary kept

Insights

Calibrate once: measure verify latency by batch, context and tree size.

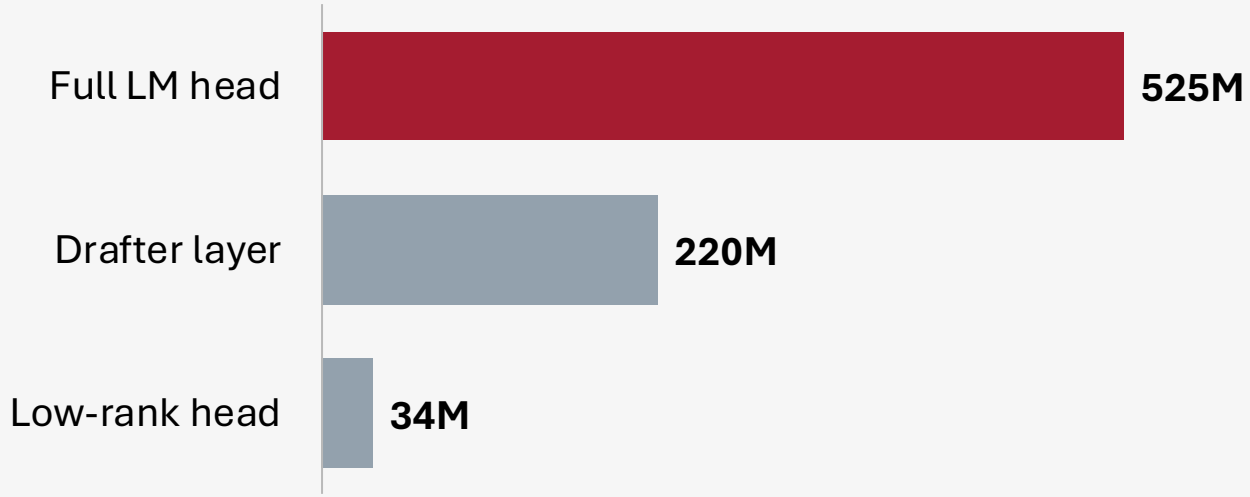
Grow by gain ÷ cost: widen or deepen while the ratio stays above a threshold.

One drafter, every load: deep trees at batch 1 or long context; chains at large batch.

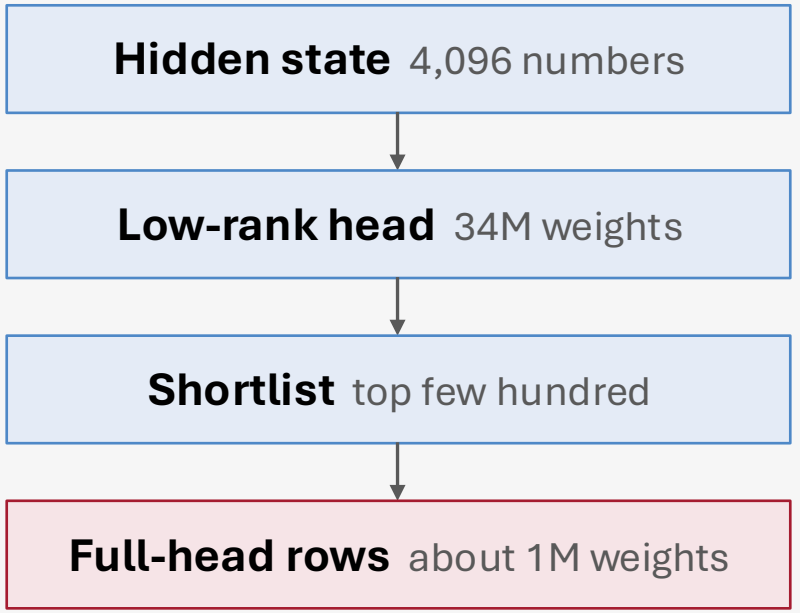
The LM head outweighs the drafter

Weights read per draft step

Llama-3-8B target: hidden 4,096 × vocabulary 128,256



Two-pass head



Insight: the drafter skips the full 128K-wide product, yet any token can still reach the shortlist.

EAGLE was widely used in production

EAGLE trains a small drafter on a frozen target's features, so a provider can add one to a model it already serves.

Meta Llama 3 and Llama 4

EAGLE in Meta's serving stack, with online distillation, tree attention and batching fixes on H100s.

1.4–2.0× faster at large batch

Baseten custom EAGLE-3 heads

Trains EAGLE-3 heads for customer models.

1.5–2.5× in production

Efficient Speculative Decoding for Llama at Scale: Challenges and Solutions
How to train custom EAGLE-3 heads for speculative decoding

DeepSeek-V3 trains its own drafter

Problem

A drafter trained after the fact must chase the target's distribution, and every new model version means training and serving a separate drafter again.

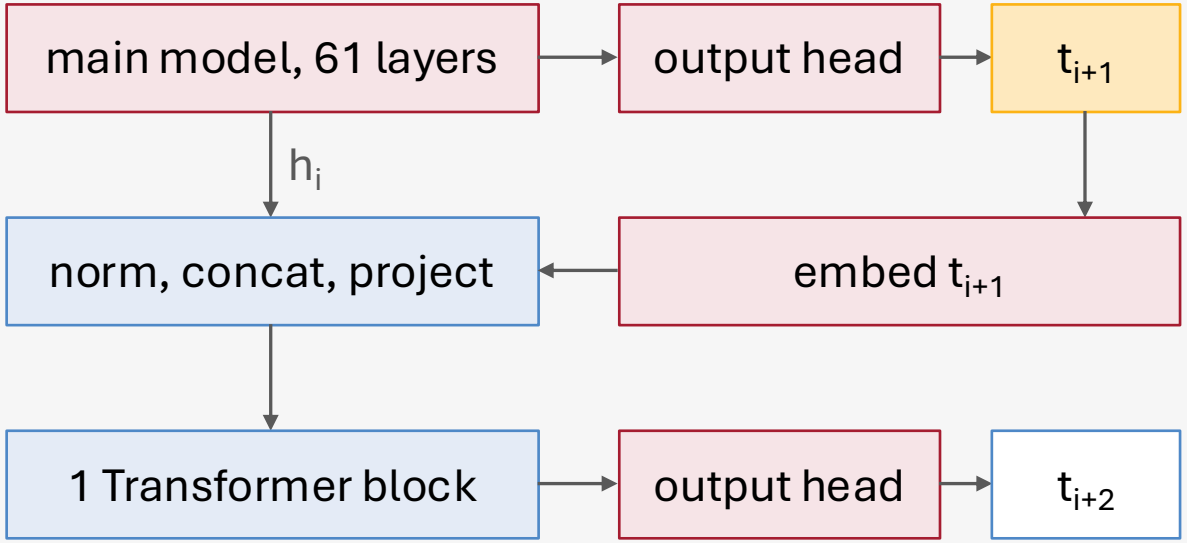
Motivation

Predicting further ahead densifies the training signal and may help planning; a jointly trained module stays aligned with the main model by design.

Key idea

One sequential MTP module (shared embedding and output head, one Transformer block) predicts token t + 2 during pre-training; at inference it serves as a k = 1 drafter.

One draft token caps MTP near 2×



Crimson: shared with the main model
Blue: new MTP weights, one layer deep

Insights

Keeps the causal chain: the module sees the true t + 1 embedding, unlike Medusa's blind heads.

Near its ceiling: with k = 1, $E = 1 + \alpha \approx 1.9$ at 85–90% acceptance; measured 1.8× tokens/s.

Trained with the model: $L = L_{main} + \lambda \cdot L_{MTP}$ ($\lambda = 0.3$, then 0.1); 14B weights beside 671B.

Question: Could a drafter propose a whole block of tokens in one pass?

Insights into DeepSeek-V3: Scaling Challenges and Reflections on Hardware for AI Architectures
DeepSeek-V3 Technical Report

DFlash drafts a block in one pass

Problem

Autoregressive drafters such as EAGLE spend k sequential passes to draft k tokens, so draft latency grows with k and caps how far ahead it pays to guess.

Motivation

Diffusion LMs fill many positions in parallel but lose quality on their own; as drafters their mistakes are caught by verification, so only their speed matters.

Key idea

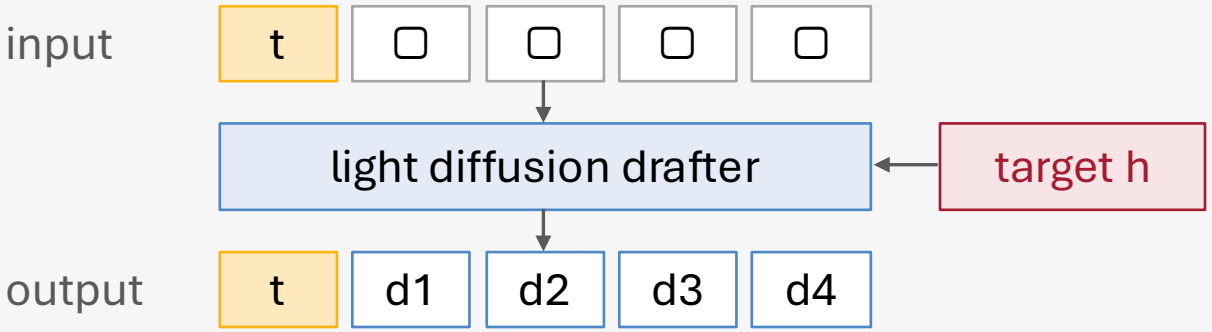
A light block-diffusion drafter, conditioned on hidden features fused from several target layers, predicts a 16-token masked block in one forward pass.

Long blocks pay off in DFlash

One round with k = 4 drafts (illustrative)



Denoise a masked block in one pass



Insights

Flat draft cost: all masked positions are filled together, so a 16-token block costs one draft pass.

Context comes from the target: draft layers attend to hidden states fused from several target layers.

Trained as it is used: random anchors, with the block after each one masked and predicted in parallel.

DSpark verifies only the likely prefix

Problem

Parallel drafters lose accuracy along the block, so verifying the full block wastes target compute that other requests need.

Motivation

Whether extra verified tokens pay off depends on each request's acceptance and on current load, so one fixed verification length is wrong for most requests.

Key idea

Add a light sequential head to DFlash's parallel backbone, a confidence head that predicts prefix survival, and a scheduler that sets each request's verify length from load.

In production, block drafters beat MTP

A block drafter fills a whole block in one pass; in production, the question is how much of each block is worth verifying under load.

DeepSeek DSpark in V4

Co-deployed with DeepSeek-V4 Flash and Pro; a confidence head sets each request's verify length under live traffic.

60–85% faster per user than MTP-1

DFlash SGLang, vLLM, TensorRT-LLM

All three engines support it; on vLLM, a DFlash checkpoint replaces EAGLE-3 with only a config change.

Up to 5.8× on Gemma 4 31B

Question: Agents often repeat text they have seen. Could we draft with no model?

DSpark: Confidence-Scheduled Speculative Decoding with Semi-Autoregressive Generation
Boost Inference Performance up to 15x on NVIDIA Blackwell Using DFlash Speculative Decoding

Retrieval-based drafts does not need a model

Problem

Agentic workloads emit long, repetitive outputs; model drafters ignore the repetition, and n-gram lookup only searches the current prompt.

Motivation

Agents repeat text across calls (code edits, SQL, tool calls), so a cache of earlier prompts and outputs can predict long continuations at almost no cost.

Key idea

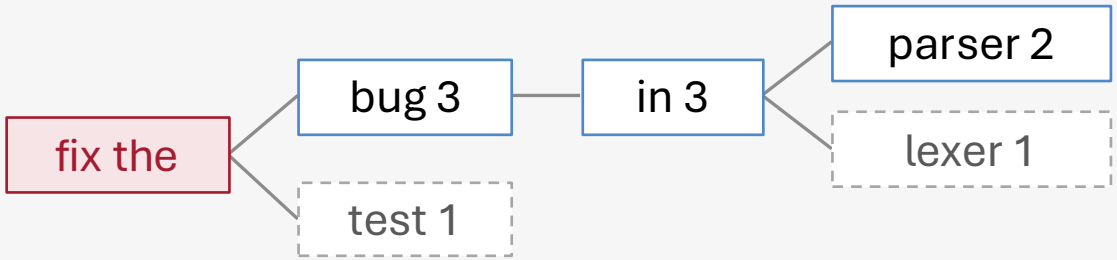
Keep suffix trees over past outputs (global) and the current request on the CPU; match the last p tokens and draft a tree of up to α·p tokens, scored by how often each continuation appeared.

Long guesses are nearly free

Past outputs, kept in a suffix tree on the CPU

"fix the bug in parser" (×2),
"fix the bug in lexer", "fix the test"

New request ends with "fix the": **match p = 2**



Blue: drafted, the highest-count paths up to **α·p = 3** tokens (α = 1.5)

Insights

Free to draft: a CPU tree lookup instead of a GPU pass, so c ≈ 0 and long guesses are affordable.

Speculate on evidence: longer matches earn longer drafts (α·p); with no match, decode normally.

History does most of the work: the global tree over past requests gives most of the gain.

Workload-dependent: up to 5.3× on agentic tasks, 2.8× faster than EAGLE-2/3 there; pair with EAGLE-3 for open chat.

In production, retrieval-based SD is common

An edit keeps most of the file, so the original is the draft: one pass verifies a long chunk, and only the changed spans decode token by token.

Cursor speculative edits

A 70B model rewrites the whole file; the current file supplies the draft, so no draft model runs.

13× faster, ~1000 tokens/s

OpenAI Predicted Outputs

API callers send the expected output, often the current file, as a prediction; matching text returns faster.

Rejected tokens are still billed

Insight: The gain scales with how much of the file survives; a full rewrite accepts little of the draft and falls back to normal decoding.

Editing Files at 1000 Tokens per Second
Predicted Outputs

Most drafters reuse the target model

Drafts come from	Family	Examples
A separate model	**Independent small model**	T5-small, Llama 3.2 1B
The target's own layers or hidden states	**Self-speculation**	LayerSkip, SWIFT
	Prediction heads	Medusa
	Feature-level drafters	EAGLE, EAGLE-2, EAGLE-3
	Native multi-token prediction	DeepSeek-V3 MTP
	Parallel and block drafters	DFlash, DSpark
Text already seen	**Copying and retrieval**	N-gram lookup, SuffixDecoding

Verification

SpecInfer verifies a tree in one pass

Problem

A single draft sequence from one small model is often wrong early, and the first rejection throws away every guess after it.

Motivation

At small batch the verify pass leaves compute idle, so checking many candidate continuations costs about as much as checking one.

Key idea

Merge small-model guesses into a token tree, verify every node in one pass with a tree mask; multi-step speculative sampling keeps the output exact.

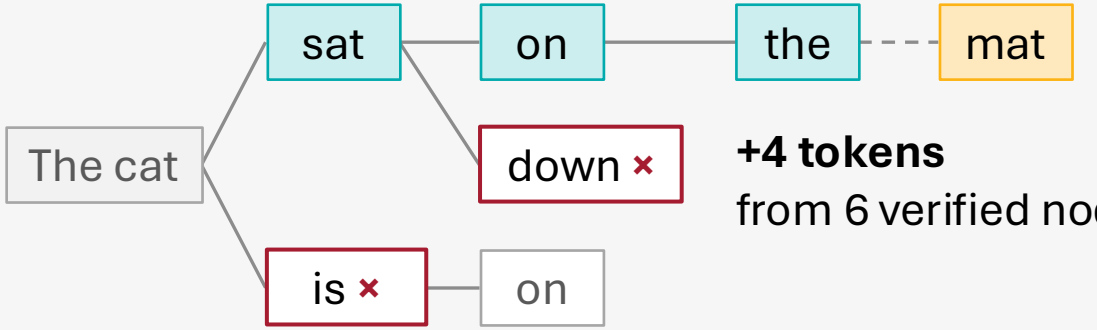
Trees yield longer accepted runs

Chain: one guess per position



+1 token

Tree: top-2 guesses per position



+4 tokens
from 6 verified nodes

accepted × rejected discarded from target

Insights

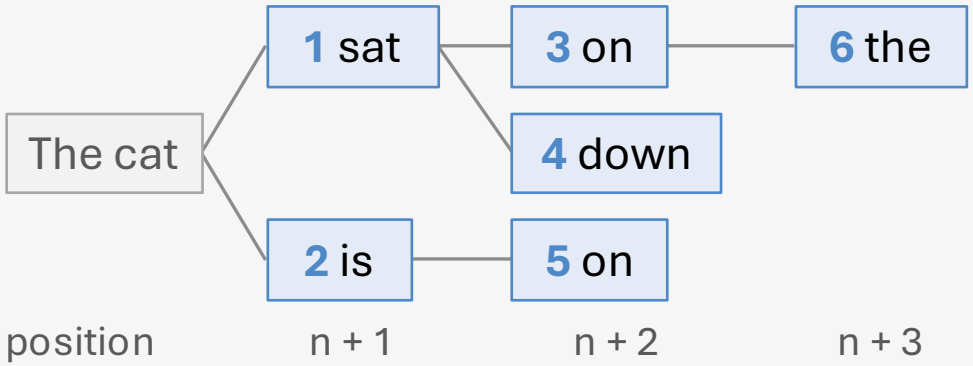
A level passes if any child does: one miss no longer ends the round.

Same pass, more tokens: at batch 1, 6 tree nodes cost the same 3.3 ms pass as 3 chain drafts (slide 15).

Any shape, one pass: nodes are flattened and masked to see only their ancestors (next slide).

Question: Each branch has its own history. How can one pass check them all?

One pass checks every branch



1 Flatten: append all nodes; position = n + depth.

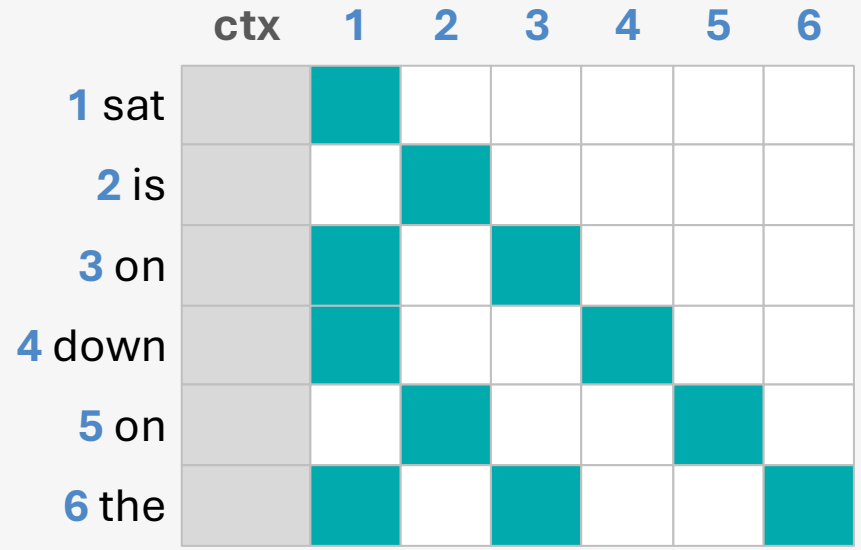
2 Mask: each node sees the context and its ancestors.

3 Score: one target pass gives p at every node.

4 Walk: keep the longest passing path, e.g. 1 → 3 → 6.

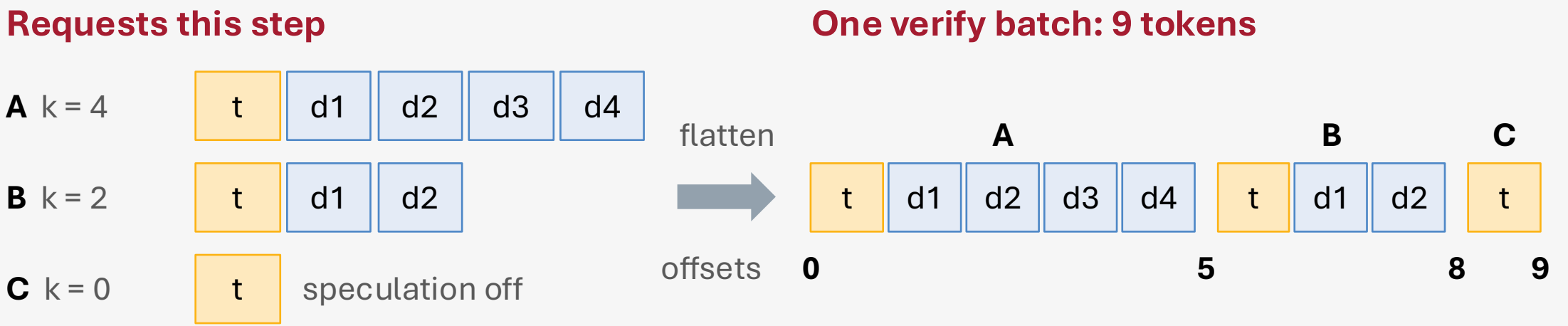
5 Compact: copy that path's KV into slots n+1..n+3.

Attention mask: row sees column



Siblings never see each other.

Ragged batches: one flat pass for all requests



Request i adds 1 + k_i query tokens; varlen attention indexes them by **per-request offsets**, rebuilt every step.

One target pass
weights read once for all 9 tokens

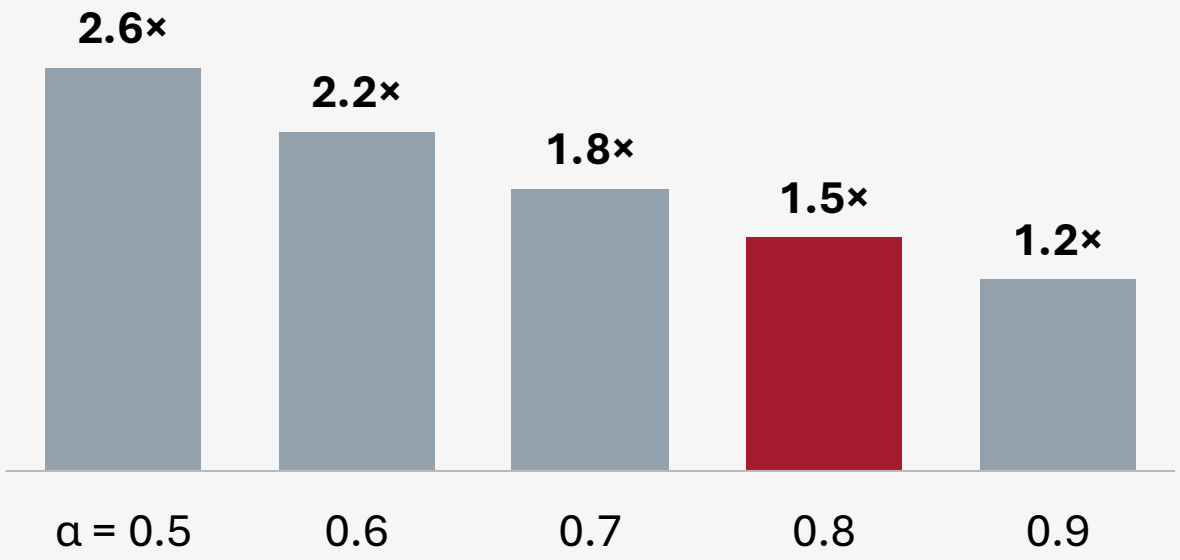
Split by offsets
A gets 5 outputs, B 3, C 1

Sample per request
accept or reject each request's own drafts

Speculation costs extra compute

Target FLOPs per kept token, vs plain decoding

k = 4 drafts per pass; drafter FLOPs not included



$$\text{cost per kept token} = \frac{k+1}{E}$$

$E = (1 - \alpha^{k+1})/(1 - \alpha)$ kept per pass

Where the waste goes

At α = 0.8, k = 4: 3.36 of 5 verified positions kept, 1.64 thrown away.

Drafter FLOPs add on top

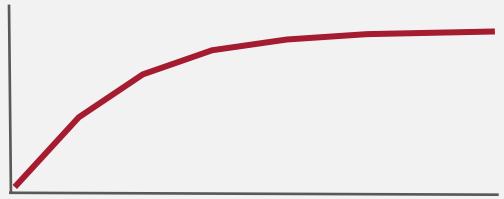
$k \cdot N_d/N_t$ target passes per round: 0.6 for 1B → 8B at k = 4.

Insight: Compute-bound, throughput scales by E/(k + 1) < 1 for any α < 1, so the best k there is 0: k must follow the load, not stay fixed.

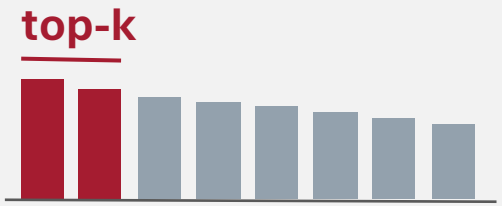
Sequoia picks the best tree shape

Problem

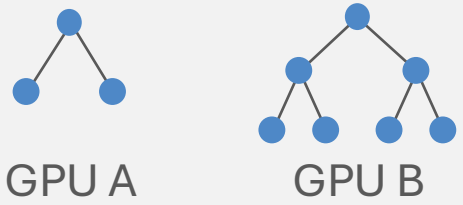
Fixed tree shapes stop gaining as trees grow, top-k sampling fails at some temperatures, and no single tree size fits every GPU.



Fixed trees plateau



Top-k misses the sample



No size fits every GPU

Motivation

Acceptance per child position can be measured once per model pair, so the best tree for a node budget is an optimization problem.

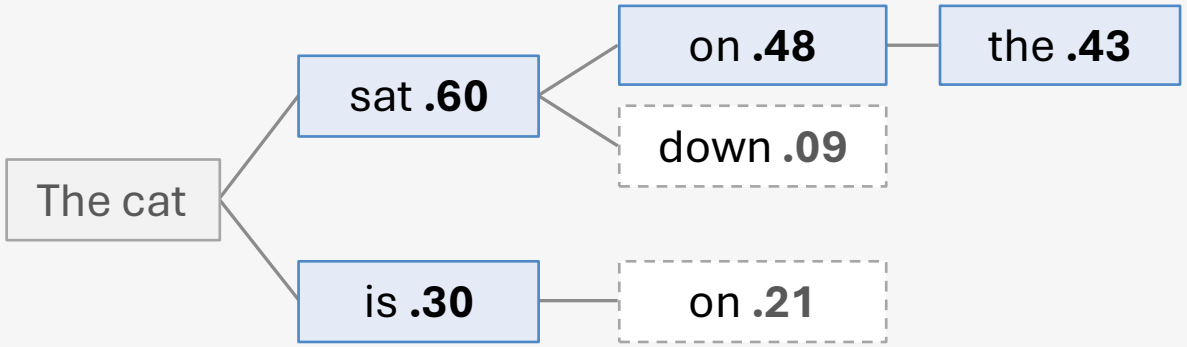
Key idea

A dynamic program builds the tree that maximizes expected accepted tokens; sampling without replacement keeps verification robust; a profiler picks tree size and depth per GPU.

Size the tree to the GPU

$$E \approx \sum_{v \in T} \prod_{u \in \text{path}(v)} c_u$$

c: drafter confidence



Blue = top 4 kept:
E ≈ .60 + .48 + .43 + .30 = 1.81 per pass

Question: Exact checks reject correct but different tokens. Could we accept them?

Insights

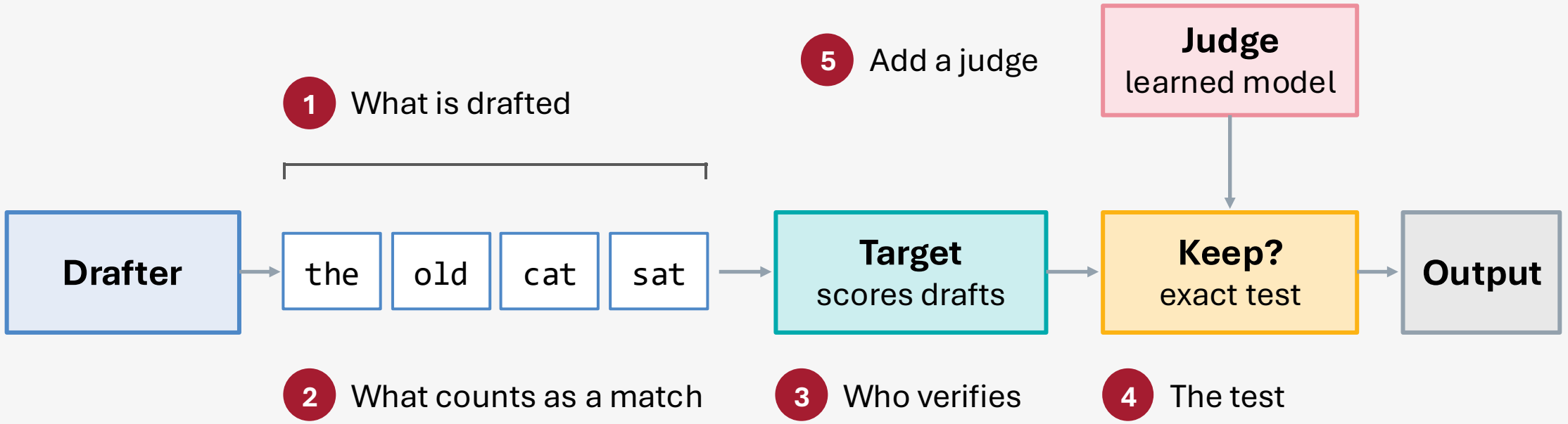
Greedy works: path values only shrink, so the top m nodes form a tree; the DP finds up to 33% more tokens.

Log returns: accepted tokens grow about logarithmically with size, but verify cost grows linearly past the knee.

In SGLang: num-steps = depth, eagle-topk = width, num-draft-tokens = m.

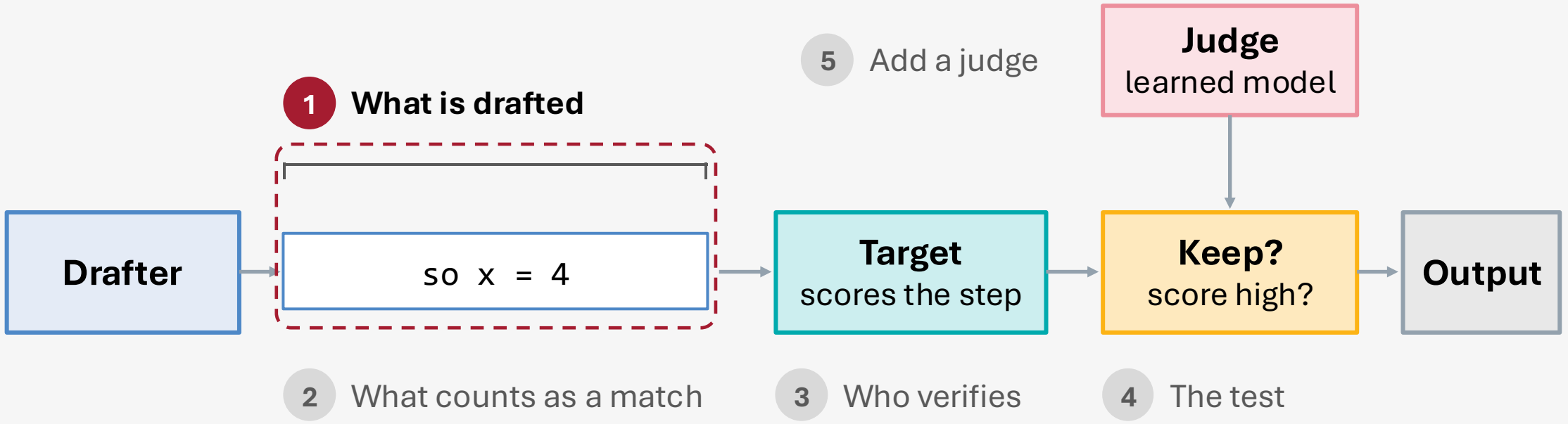
Sequoia: Scalable, Robust, and Hardware-aware Speculative Decoding
EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees
Speculative Decoding, SGLang Documentation

Lossy speculative decoding: relax the loop



Exact SD drafts tokens, verifies them against the full target. Each numbered point is one way to accept more drafts, and each one changes the output distribution.

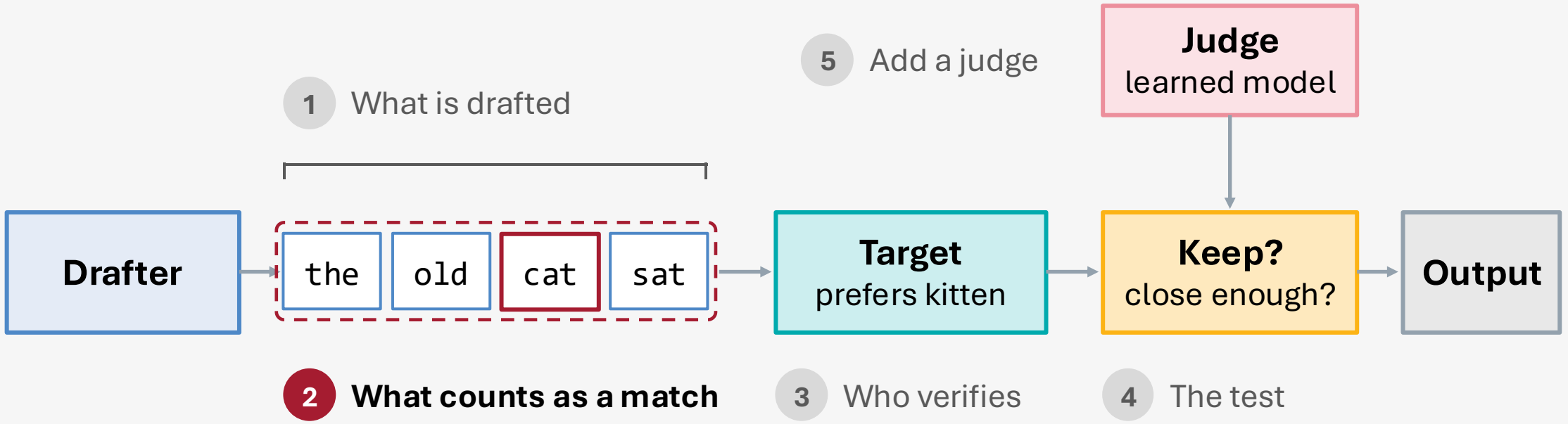
Draft whole reasoning steps



Idea: the drafter writes a whole reasoning step. The target or a judge scores it, and only low-scoring steps are regenerated.

Methods: SpecReason, Lookahead Reasoning, ConfSpec, SPEC-RL

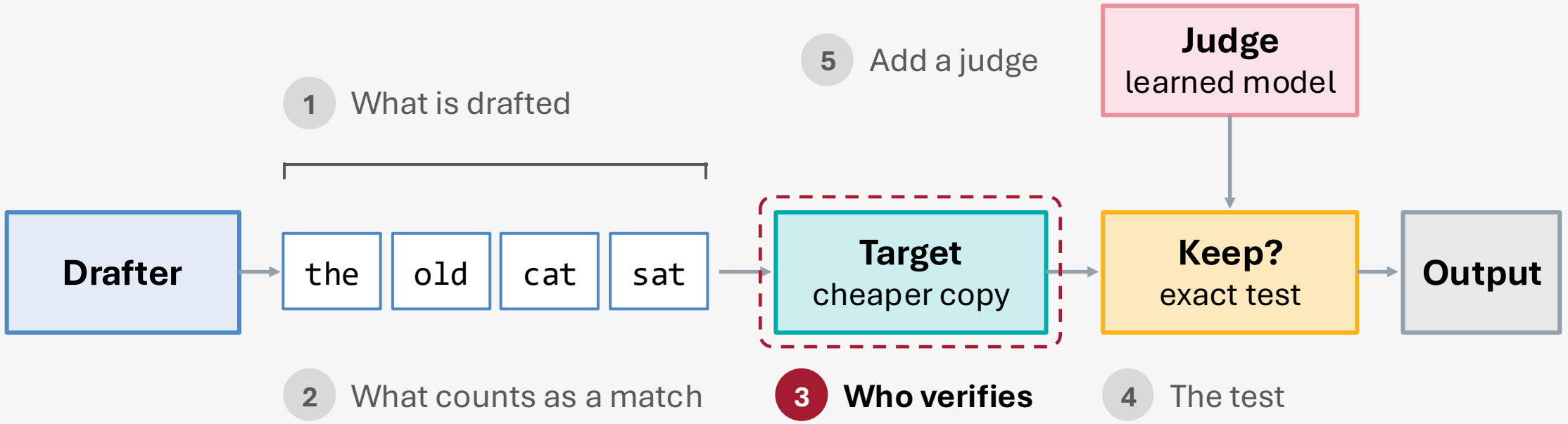
Accept near-duplicate tokens



Idea: keep a draft token that is close to the target's choice, such as "cat" when the target prefers "kitten", or a neighbouring image token.

Methods: LANTERN, grouped SD, speech SSD, Spec-VLA

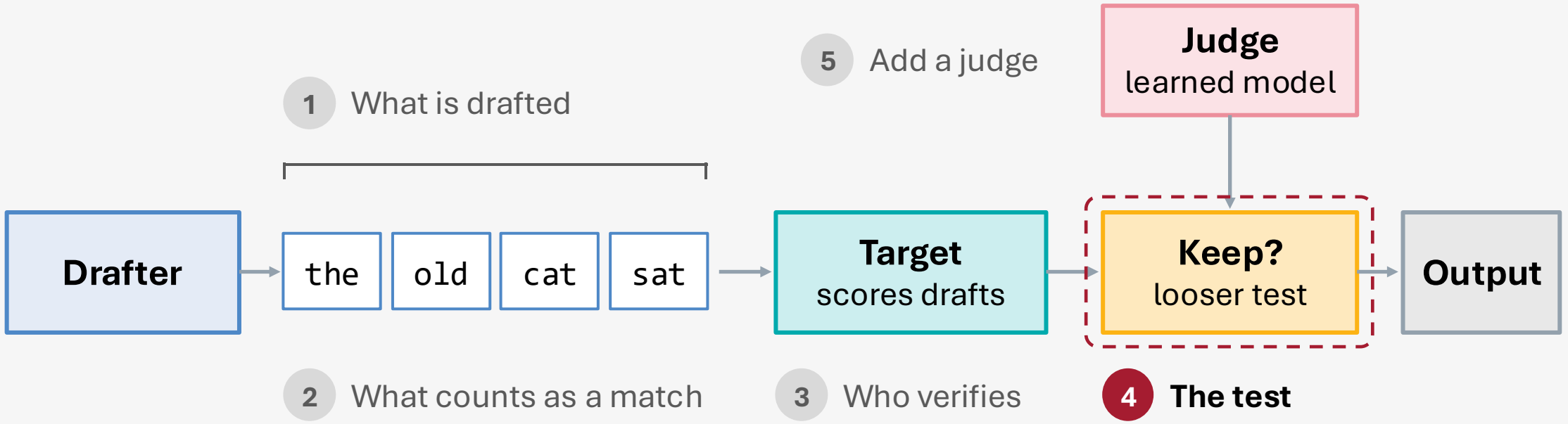
Verify with a cheaper target



Idea: check drafts against a cheaper copy of the target, with a sparse KV cache or fewer active experts. Verification costs less, but the output drifts from the full model.

Methods: Dustin, MoE-Spec

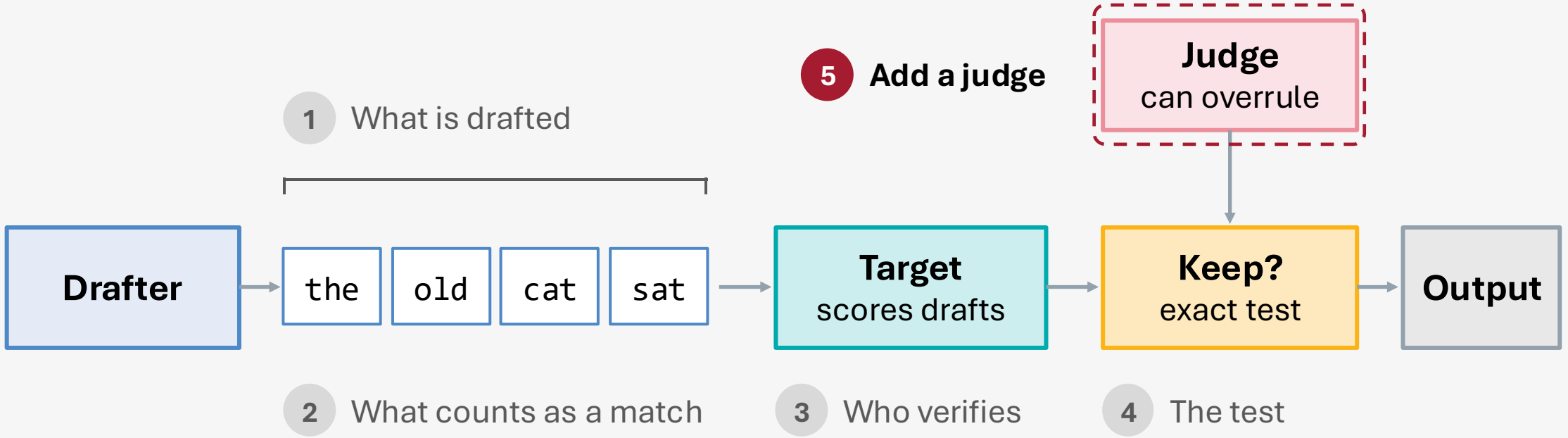
Loosen the acceptance test



Idea: replace the exact accept rule with a looser one: keep any draft token the target finds likely enough, even when the exact test would reject it.

Methods: Lenience, top-β, typical acceptance

Let a judge keep good rejections



Idea: a small classifier reads the target's hidden state at a rejected token and keeps the token when the mismatch would not hurt the answer.

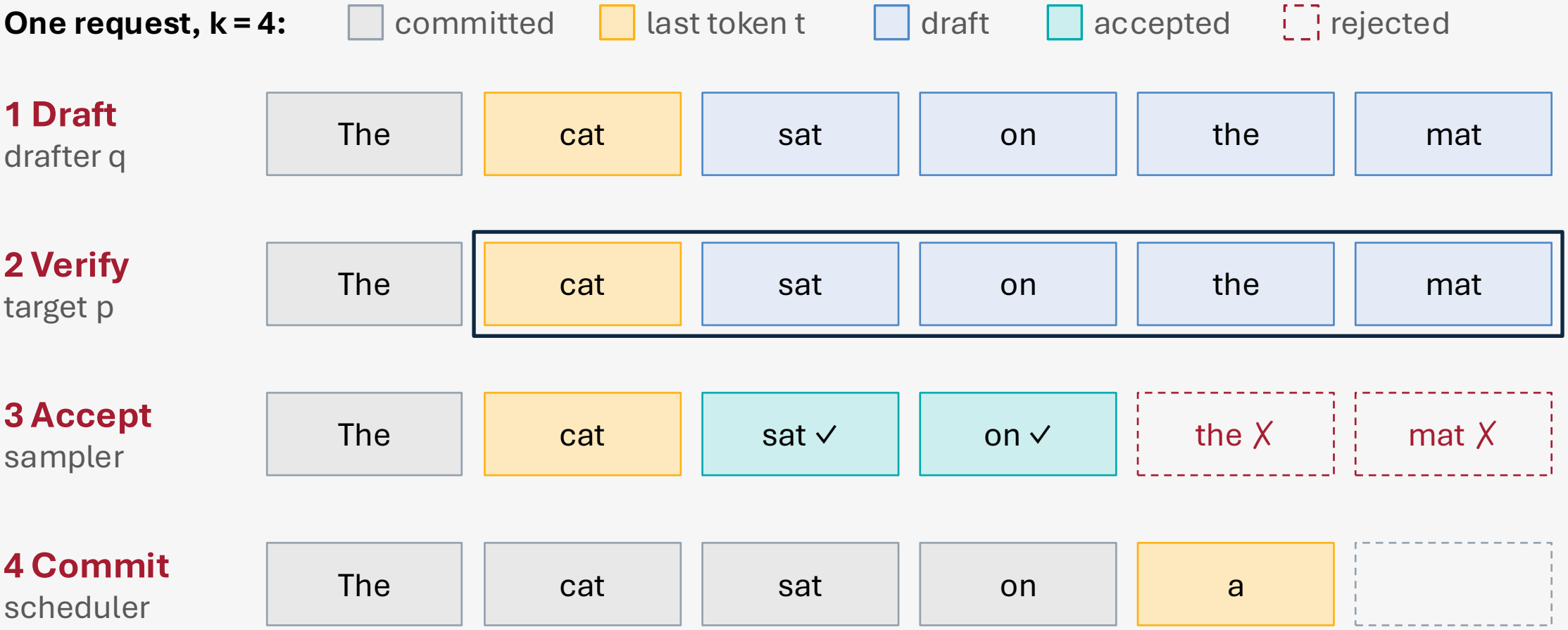
Methods: Judge Decoding, AutoJudge, SelfJudge, FLy

Lossy SD relaxations: five places to accept more

1	**What is drafted**	Draft whole reasoning steps, scored by a reward model or judge SpecReason, Lookahead Reasoning, ConfSpec, SPEC-RL
2	**What counts as a match**	Accept a near-duplicate token close to the target's choice LANTERN, grouped SD, speech SSD, Spec-VLA
3	**Who verifies**	Verify with a cheaper target: sparse KV or fewer experts Dustin, MoE-Spec
4	**The test**	Loosen the acceptance rule itself Lenience, top-β, typical acceptance
5	**Add a judge**	A small classifier overrides some rejections Judge Decoding, AutoJudge, SelfJudge, FLy

Optimizing Speculative Decoding in Systems

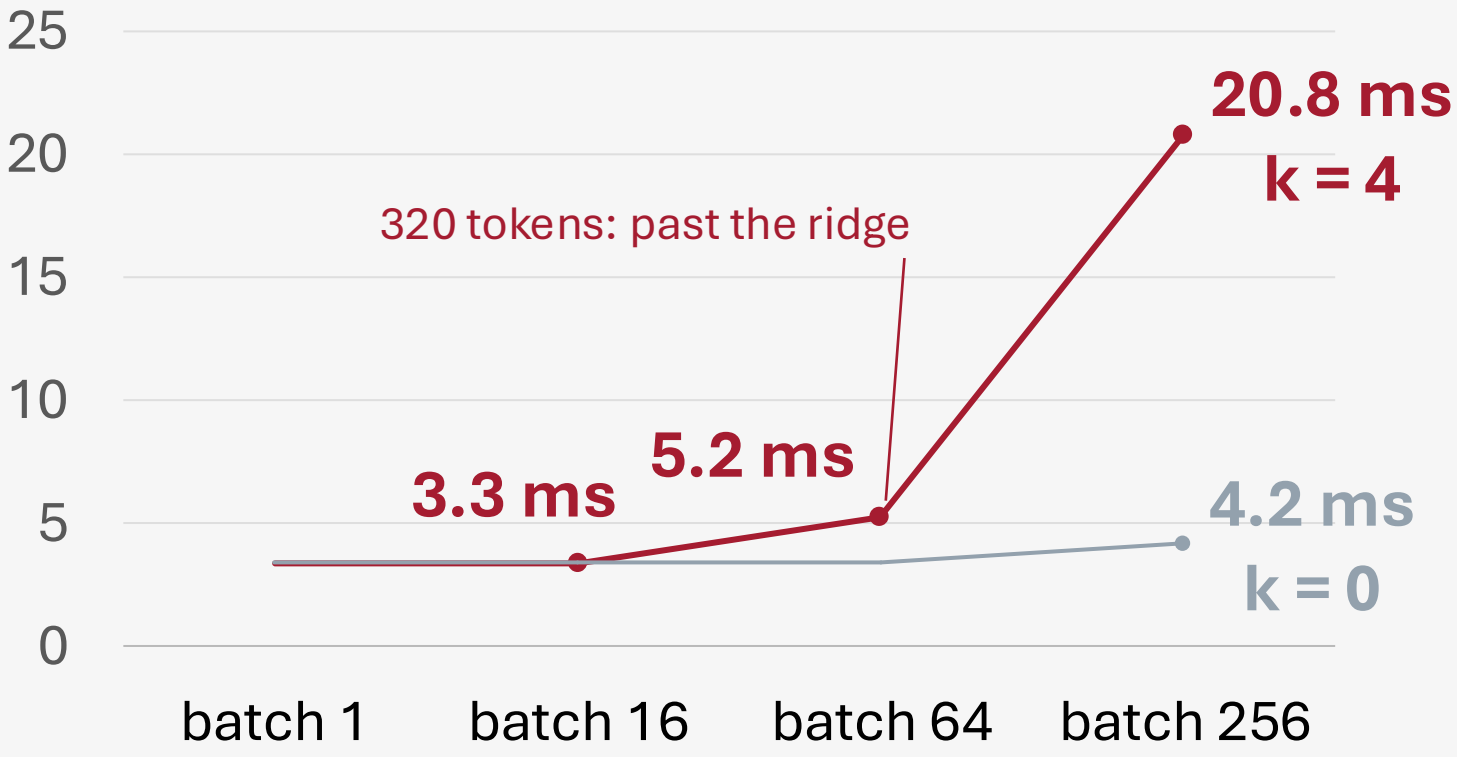
Speculation fits in one engine step



Inside vLLM: Anatomy of a High-Throughput LLM Inference System
Consistent semantics for drafted, verified and accepted spec decode tokens

Step time: drafts cost nothing until the GPU is busy

Step time per verify pass, ms, by batch size
ideal H200 roofline, Llama-3-8B FP16, weights only



Flat up to ~206 tokens: each pass waits on the 16 GB weight read, so 80 tokens cost what 16 do.

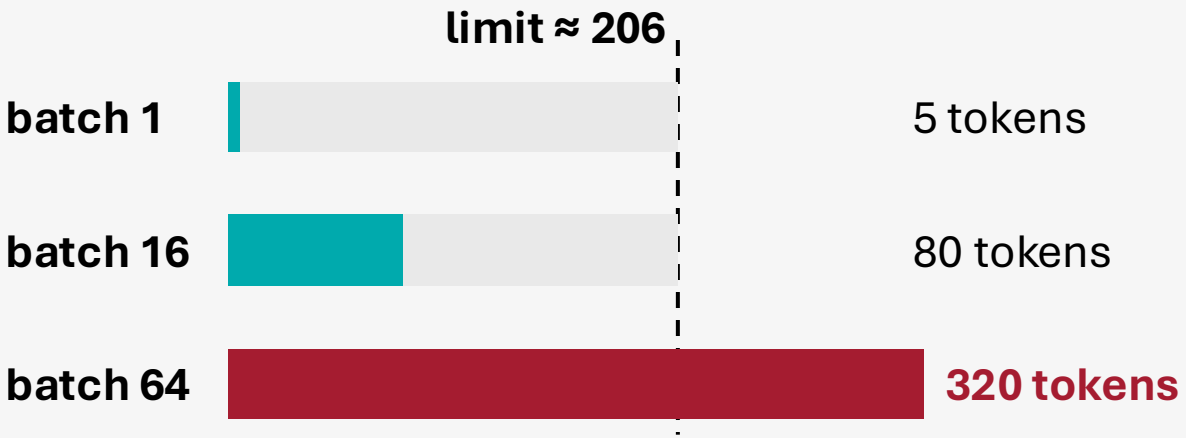
Past it, every token adds time: about 0.016 ms each, so 1,280 tokens take 20.8 ms.

NVIDIA, 2024: H200 Tensor Core GPU datasheet (4.8 TB/s HBM3e, 989 TFLOPS dense BF16)

Large batches erase the free compute

Tokens per weight read in one verify pass, k = 4

H200 needs ~206 to stay busy (slide 5); weights only



Below the limit, 5 tokens cost about as much as 1; above it, every token adds time.

Per-user vs total speed

Batch 16: k = 4 lifts each user from 299 to 1,005 tok/s at the same 3.3 ms step.

Batch 64: 320 tokens per pass, 5.2 ms: 647 tok/s per user, 41K total.

Batch 256: k = 4 stretches the step to 20.8 ms: 162 tok/s per user and 41K total, below k = 0's 241 and 62K.

Ideal H200 roofline, α = 0.8, weights only, drafter free.

Insight: Batching and speculation spend the same ~206 free slots: batching buys total throughput, speculation buys per-user speed. Splitting them is an SLO choice.

TurboSpec picks k every step

Problem

Under continuous batching, speculation can raise latency: at high request rates or low acceptance, verifying rejected drafts takes compute from other requests.

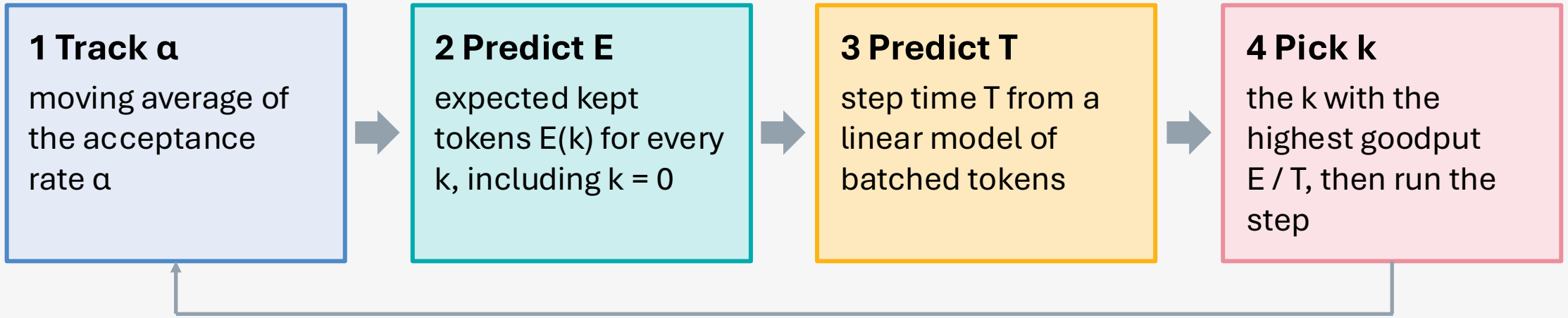
Motivation

The best k depends on batch size and acceptance, which change every step, but both the accepted length and the batched step time are predictable.

Key idea

Goodput = generated tokens ÷ step time. Predict each request's accepted length from its acceptance history and the step time with a linear model; pick each k to maximize goodput.

TurboSpec runs a goodput loop every step



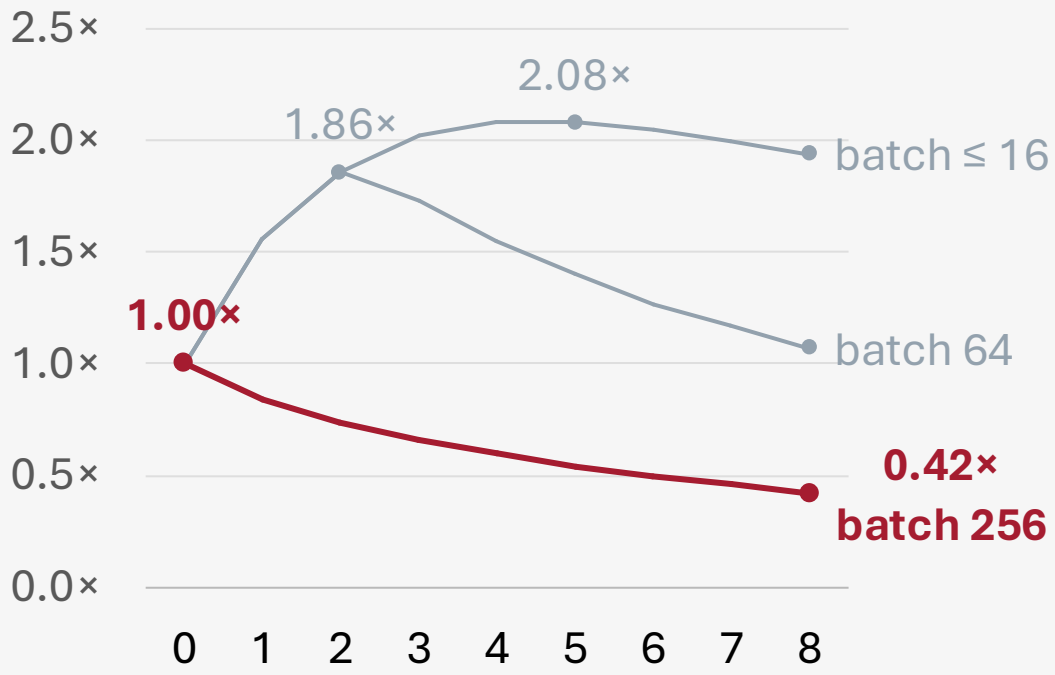
What it picks (α = 0.8):

batch ≤ 16: k ≈ 5 | **batch 64: k ≈ 2** | **batch 256: k = 0**

The best k shrinks as batches grow

Goodput vs plain decoding, by drafts per step k

Model: ideal H200 roofline, Llama-3-8B, 1B drafter, α = 0.8



Question: Large batches erase the gain. Can anything bring speculation back?

Insights

Optimize goodput, not acceptance: $\Sigma E_i(k_i) \div$ step time; E from acceptance history, step time from a linear model.

The best k shrinks with load: about 5 up to batch 16, 2 at batch 64, 0 at 256 in this model; SmartSpec saw > 4 at small batch.

Drafting has a cost: each extra draft adds a drafter pass (0.52 ms for a 1B model here), so even small batches peak.

Never worse than off: at peak load it proposes nothing; at light load, up to 3.2× lower latency.

Long context revives speculation

Problem Speculation is thought to pay off only at small batch sizes.

Motivation At long context, KV reads dominate, so large batches stay memory-bound.

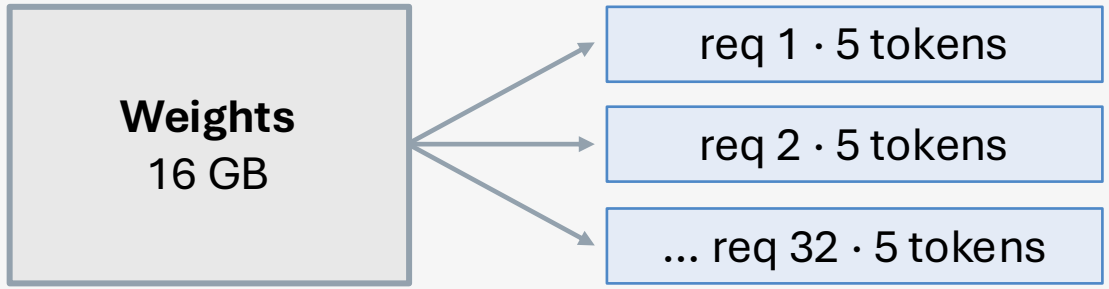
Key idea Draft on a fixed-size sparse KV cache; verify on the full cache.



MagicDec: Breaking the Latency-Throughput Tradeoff for Long Context Generation with Speculative Decoding

KV reads are private to each request

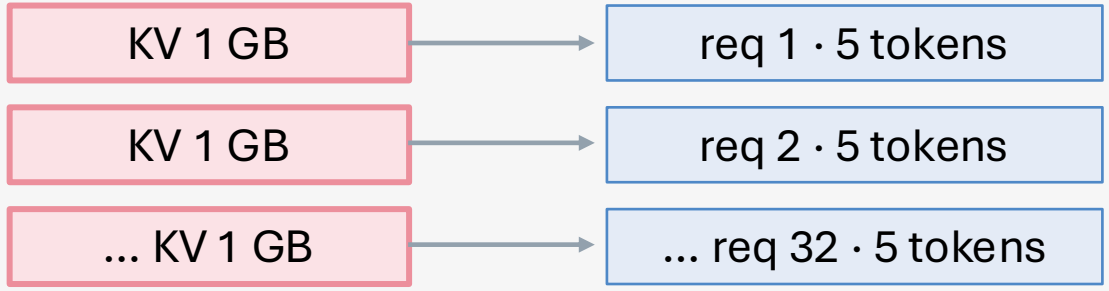
Weights: one read serves the whole batch



160 FLOPs per byte

32 requests × 5 tokens share one weight read, so batching raises this toward the 206 an H200 needs to stay busy.

KV cache: each request reads its own



20 FLOPs per byte

Each 1 GB at 8K context serves only its own 5 tokens (4 FLOPs per byte each with GQA), far below 206 at any batch size.

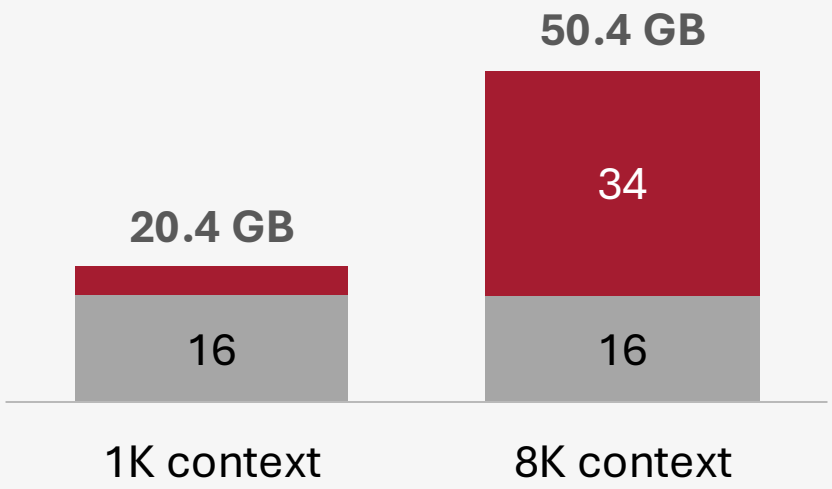
MagicDec: Breaking the Latency-Throughput Tradeoff for Long Context Generation with Speculative Decoding

Speedup grows with batch at long context

Bytes read per decode step, GB

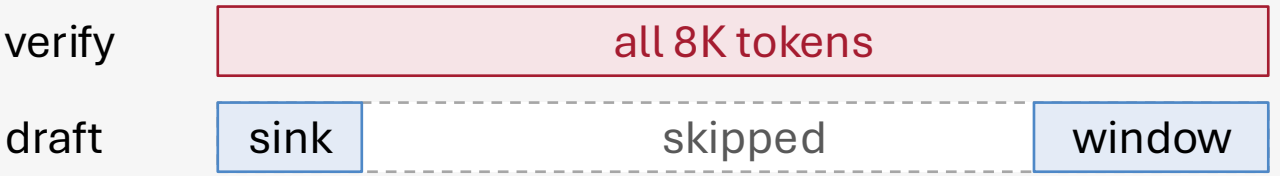
Llama-3-8B, batch 32, one H200

gray: **weights** · red: **KV cache**



Question: Is verifying k tokens still cheap for MoE models?

KV read per pass at 8K context



Draft cost stays flat in context; the drafter can even be the target

Insights

KV reads aren't shared: batch 32 at 8K reads 34 GB of KV vs 16 GB of weights per step.

Attention stays memory-bound: ~4 FLOPs per KV byte with GQA, 4(k + 1) with k drafts, ≪ 206.

Gains grow with batch past ~4K tokens: up to 2.51× at batch 32–256 (Llama-3.1-8B).

MagicDec: Breaking the Latency-Throughput Tradeoff for Long Context Generation with Speculative Decoding

Cascade turns speculation off for MoE

Problem

On MoE models each verified token routes to its own experts, so verifying k drafts reads many more weights; always-on speculation slowed requests by up to 1.5×.

Motivation

The payoff varies by model and task (Mixtral, k = 3: 39% faster on code, 54% slower on math) but stays stable over short windows of a request.

Key idea

Utility = token gain ÷ verification cost. Per request, test a few k briefly, then run the best k for a longer phase, or turn speculation off when utility falls below 1.

Short tests limit MoE slowdown

One Mixtral layer, k = 7: 8 experts, 2 per token

decode 1 token



2 experts

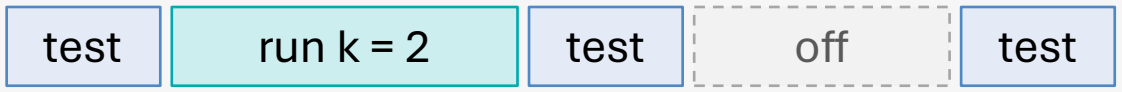
verify 8 tokens



≈7 experts

$N(1 - (1 - k/N)^n) \approx 7.2$ of 8 (3.6×); ~3× measured, as nearby tokens share experts.

Cascade on one request



short tests, longer set phases; utility < 1 turns speculation off

Insights

Utility is stable over short windows: a brief test predicts the next, longer phase.

Off still costs a little: the drafter keeps running so its KV stays current (2–3%).

Finer experts hurt more: 8 tokens read 7.2× the routed experts in DeepSeek-V3 vs 3.6× in Mixtral (uniform routing).

Result: worst-case slowdown 5% instead of 1.5×; 7–14% more throughput than static k.

Decide k at run time

Goodput = $\Sigma_i E_i(k_i) / T(\Sigma_i (1 + k_i))$: choose each k_i to maximize it.
E: expected kept tokens for request i · T: step time for that many batch tokens

SmartSpec batch size

E from acceptance history, T from a linear step-time model; at peak load it picks k = 0.

Up to 3.2× lower latency

Cascade MoE

Tests k in short phases; utility below 1 turns speculation off, but the drafter keeps running so its KV stays current (2–3%).

Worst case: 5% slower

DSpark production

A confidence head estimates prefix survival; the scheduler sets each request's verify length from load profiles.

60–85% faster per user

Insight: All three maximize the same goodput and differ only in how they estimate E: acceptance history, short trials, or a learned confidence head.

DSpark: Confidence-Scheduled Speculative Decoding with Semi-Autoregressive Generation
Optimizing Speculative Decoding for Serving Large Language Models Using Goodput
Utility-Driven Speculative Decoding for Mixture-of-Experts

Every idea pulls one of three levers

$$S = \frac{E}{ck + 1}$$

Raise E

Accept more tokens per pass

Train drafters to agree
DistillSpec, EAGLE-3

Try a tree of guesses
SpecInfer, EAGLE-2, Sequoia

Accept near-misses
Judge Decoding

Shrink c·k

Make drafting cheaper

Reuse the target model
LayerSkip, Medusa, EAGLE, MTP

Draft all k in one pass
DFlash, DSpark

Copy text already seen
N-gram lookup, SuffixDecoding

Keep the 1 at 1

Keep verification at one step

Size k to the load
SmartSpec, EAGLE-4

Speculate at long context
MagicDec

Switch it off for MoE
Cascade

Verify the likely prefix
DSpark

Next class

- **Mon, Nov 16:** Efficient LLM serving: KV-cache optimization and model architectures (student-led)
- **Wed, Nov 18:** Student sharing III: showcase your serving engine (Assignment 4)
- **Assignment 5** (Mads-Stack: optimize the full stack) is out today; due Tue, Nov 24, 11:59 pm

$$S = \frac{E}{ck + 1}$$

Optional topics

- Lossy speculative decoding