EFFICIENT EAGLE-3 TRAINING VIA STRUCTURE-AWARE EXECUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Speculative decoding speeds up large language model inference using a lightweight drafter to propose multiple candidate tokens which the target model checks in parallel. EAGLE-3 (Li et al., 2025) is a recent draft-model architecture for speculative decoding. The data used to train drafters have heterogeneous supervision ratios. That is, the fraction of valid token positions with direct supervision varies across samples and datasets. Existing EAGLE-3 training frameworks still execute recurrent computations that cannot affect the training loss and recompute token-side projections for inputs shared by different recurrent paths, incurring avoidable computation and memory costs. We propose structure-aware execution based on two observations: some recurrent updates cannot affect a supervised prediction, and different computations can repeatedly project the same token-side input. These observations lead to three execution changes. First, we keep only recurrent computations needed by selected losses at the current or later training-time test (TTT) steps, while retaining their same-path predecessors and the full public key/value (K/V) context required by the retained queries. Second, we design fused attention to process these potentially non-contiguous queries without storing public-only outputs and log-sum-exp (LSE) values for a later merge. Third, we compute and reuse a shared token-side partial for each source position while separately projecting each computation's hidden state, avoiding repeated token-side projection work. On CNN and ShareGPT, our method achieves a 1.85×–4.34× end-to-end training speedup over four existing training frameworks. Numerical checks show close agreement with SpecForge, and the trained drafters maintain comparable draft acceptance on held-out test examples.

1 INTRODUCTION

Speculative decoding uses a small drafter to propose tokens, which the target model then verifies (Leviathan et al., 2023). EAGLE-3 (Li et al., 2025) is a recent draft-model architecture that reuses features from multiple layers of the target model to generate candidate tokens. To work well, the drafter needs to learn what the target model is likely to generate for the actual tasks. Therefore, the drafter's training data should reflect both the target model's outputs and these tasks. Online Speculative Decoding and Aurora study how to adapt drafters to queries received during use (Liu et al., 2023; Wang et al., 2026a). Training and updating these task-specific drafters can be costly, especially when training uses long sequences and runs the drafter for several recurrent steps.

We study four real-world datasets for drafter training: ShareGPT-20k, CNN, SQL, and DRAgent. ShareGPT-20k (Chiang et al., 2023) is one source of prompts used in the original EAGLE-3 training setup (Li et al., 2025). We construct the CNN and SQL datasets following the task-specific setup used by HedgeSpec (Liu et al., 2026b) for EAGLE-3 training. For DRAgent, we generate agent traces from questions in DeepResearch Bench (Du et al., 2025) and DeepResearchGym (Rosset et al., 2024; Coelho et al., 2025), using a workflow based on Open Deep Research.[1] Figure 1 shows the supervision ratio of each sample. This is the fraction of valid token positions that receive direct supervision. The ratio varies widely across samples and workloads. Only selected assistant-output tokens receive direct supervision. System instructions, user prompts, tool results, and other

[1] Open Deep Research: `https://github.com/langchain-ai/open_deep_research`.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

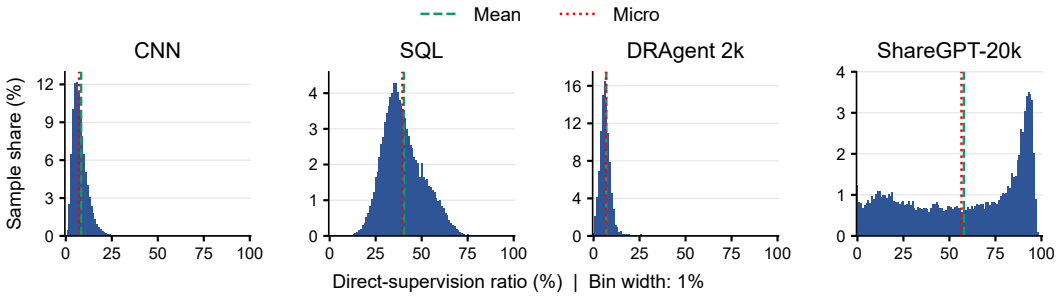

Figure 1: **Distributions of direct-supervision ratios.** Green dashed lines show the mean ratio across samples, with equal weight per sample. Red dotted lines show total supervised positions divided by total valid positions.

unselected tokens remain in the sequence but receive no direct supervision. Appendix D.1 explains how task templates, interaction structure, and tool outputs lead to these differences.

The supervision ratios in Figure 1 suggest opportunities to skip work, but the supervision mask alone cannot identify which computations to skip. EAGLE-3 trains its drafter over several recurrent steps, called training-time test (TTT). During EAGLE-3 training, even if one step on a TTT path has no direct loss, a later step on the same path may still receive supervision. Moreover, even when a recurrent update can be skipped, public key/value (K/V) from that position may still be needed by other paths. These two roles require specialized execution rules. Besides, during EAGLE-3 training, the computations that remain can also share work. Different TTT paths and TTT steps may read the token-side input at the same source position. Their hidden states generally differ, but the token-side part of the Q/K/V projection is the same and can be computed once. In the existing EAGLE-3 training frameworks, AngelSpec, SpecForge, TorchSpec, and Speculators still execute recurrent work that cannot reach a selected loss and repeat token-side projections at each computation (Liu et al., 2026a; Li et al., 2026; TorchSpec team & Mooncake team, 2026; Speculators contributors, 2026).

In this paper, we propose Structure-Aware Execution to utilize these loss dependencies and shared inputs during training. First, we retain only the recurrent computations needed by selected losses at the current or later TTT steps, including their same-path predecessors, while preserving full public K/V. This removes recurrent work that cannot reach a selected loss. Second, we use fused attention to process the potentially non-contiguous retained queries directly, avoiding storage of public-only outputs and log-sum-exp (LSE) values for a later merge. Third, we compute and reuse a shared token-side partial for each source position while projecting each hidden state separately. This avoids repeated token-side projection work while accumulating gradients from every use of the shared partial. In summary, our contributions are:

- **Loss dependencies and shared inputs.** We identify necessary recurrent work in real-world training workloads and the shared projection inputs during EAGLE-3 training. Loss-Reachable Recurrent Execution (C1) skips unnecessary recurrent work while retaining required same-path predecessors and full TTT-step-1 public K/V.
- **Efficient execution of the retained work.** Fused Attention for Retained Queries (C2) fuses public and private attention for C1's compact queries, avoiding public-only outputs and log-sum-exp (LSE) values for a later merge. Shared-Source Projection Reuse (C3) computes token-side projections once per source position while separately projecting each hidden state.
- **Efficiency and training fidelity.** Compared with four external baselines on CNN and ShareGPT training workloads, our method reduces peak PyTorch allocated memory by 29.05%–54.42% and achieves a 1.85×–4.34× end-to-end training speedup. Separate numerical checks on DRAgent and ShareGPT yield a mean all-gradient relative-L_2 error of at most 0.62% against SpecForge. The trained checkpoints also maintain comparable draft acceptance on held-out test examples.

2 Related Work

2.1 Drafter Training and Domain Alignment

Prior work improves drafter quality by aligning training with the target model, decoding process, deployment queries, or task domain. EAGLE-3, HASS, DistillSpec, domain-draft training, Online Speculative Decoding, and Aurora represent these directions (Li et al., 2025; Zhang et al., 2024; Zhou et al., 2023; Hong et al., 2025; Liu et al., 2023; Wang et al., 2026a). These methods motivate training drafters on task-specific data. We study how to execute this training efficiently while keeping its supervision and objective.

2.2 Training Infrastructure and Efficiency

SpecForge, TorchSpec, and Speculators provide drafter-training systems (Li et al., 2026; TorchSpec team & Mooncake team, 2026; Speculators contributors, 2026). Our SpecForge baseline limits its output-head and loss computation to selected positions. C1 also restricts recurrent execution, preserving required predecessors and full public K/V. PARD and P-EAGLE reduce parallel-drafter training work through token dropping and sequence partitioning (An et al., 2025; Hui et al., 2026). P-EAGLE targets a parallel-prediction drafter and preserves attention dependencies across prediction depths (Hui et al., 2026). These parallel attention dependencies differ from the recurrent, same-path state dependencies in EAGLE-3's sequential TTT unroll. C1 derives the computations required by a fixed supervision set, retaining their same-path predecessors and the full TTT-step-1 public K/V while skipping loss-unreachable work. In AngelSpec's EAGLE-3 implementation, public attention uses full-sequence queries, and its outputs and LSEs are stored for a subsequent merge with private attention (Liu et al., 2026a; Tencent, 2026). C2 instead processes C1's compact queries and combines public and private attention in one forward kernel, avoiding those public-only intermediate results. Appendix B.1 compares C2 with an AngelSpec-derived reference adapted to the same retained queries.

3 Background and Workload Structure

At inference, EAGLE-3 fuses the target LLM's features with the current-token embedding, predicts the next draft token, and carries the predicted token and drafter state into the next draft step before target verification (Li et al., 2025). Training simulates this process with a k-step training-time-test (TTT) unroll (Figure 2). We use one-based indices throughout the paper. A sequence contains L positions, indexed by $\boldsymbol{F} = \{1, \ldots, L\}$. We use $\boldsymbol{S} \subseteq \boldsymbol{F}$ to denote the set of positions that receive direct supervision. A ***TTT path*** starts from an original sequence position r and runs the recurrent drafter for up to k prediction steps. A ***TTT step*** $j \in \{1, \ldots, k\}$ is the j-th prediction step along that path. We use (r, j) to denote the drafter computation at TTT step j on the path starting at position r.

The target LLM follows the usual autoregressive shift: computation at position a predicts the token at sequence position $a + 1$. For each position $a \in \boldsymbol{F}$, f_a denotes the target-derived feature vector assembled from the frozen target LLM's selected hidden-layer outputs at its computation position a. The ordered sequence $(f_1, \ldots, f_L)$ supplies the full target features. Let x_a denote the a-th serialized training token, and let $\tilde{x}_a$ denote the corresponding target-predicted token. Separately, e_a denotes the token-side representation of x_a after the prescribed embedding and input preprocessing. During training, the drafter reads x_a rather than $\tilde{x}_a$. Under the autoregressive shift above, the a-th token is predicted from computation position $a - 1$. Therefore, a indexes the predicted token rather than the input position for $\tilde{x}_a$. At computation (r, j), the drafter reads token-side input e_{r+j} and hidden-side input $h_{r,j-1}$, and produces state $h_{r,j}$. For position r, the computation takes f_r as input when $j = 1$ and otherwise takes the preceding hidden state $h_{r,j-1}$ from the same TTT path. Applying the LM head to $h_{r,j}$ produces draft distribution $\hat{p}_{r,j}$ for the token at position $r + j + 1$, whose soft teacher distribution is $\tilde{p}_{r+j+1}$. Indexing the draft distribution by (r, j) distinguishes different TTT paths that predict the same token position. Throughout this paper, public and private K/V refer to the drafter's attention K/V, not the target LLM's KV cache. The target LLM supplies the target-derived features f_a and teacher distributions $\tilde{p}_a$, while the drafter constructs its own K/V inside its attention layers. The drafter's public K/V form a full-sequence bank produced at TTT step 1 and shared by all TTT paths, with each query seeing only the causal prefix allowed by its attention mask. There is

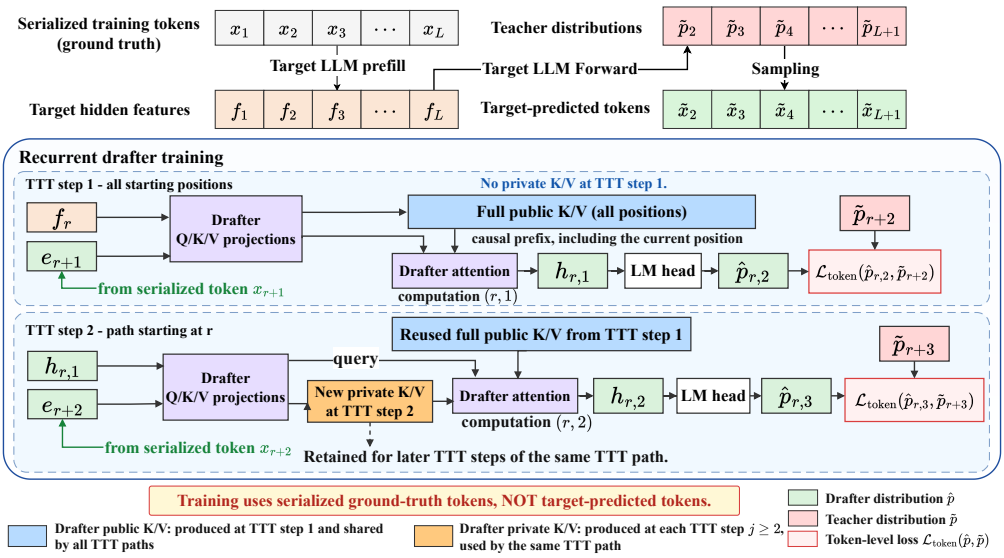

Figure 2: **Two steps of EAGLE-3 training-time test.** Later TTT steps add one new same-path private K/V entry to the same TTT path, and attend to all accumulated K/V entries.

no private K/V at TTT step 1. For $j \geq 2$, computation (r, j) reuses that public bank and appends one same-path private K/V entry, visible only along the TTT path starting at r. Thus, computation $(r, 1)$ uses $h_{r,0}$ derived from f_r and token-side input e_{r+1}, produces $h_{r,1}$, and predicts the token at position $r+2$. Appendix A, especially Table 5, gives the complete notation and execution domains.

4 Structure-Aware Execution

Figure 3 summarizes our design. We use loss dependencies to decide which recurrent computations to execute and shared token-side inputs to decide which projections to reuse. Specifically, C1 retains the computations needed for current or future losses while keeping the full TTT-step-1 public K/V. For C2, we design a fused attention directly for the resulting compact queries without materializing the public-only outputs and log-sum-exp (LSE) values for a later merge. C3 shares token-side projections across computations that read the same source position. We introduce each component in detail in this section and provide implementation notes in the Sections A.1, B, and C of the appendix.

4.1 Loss-Reachable Recurrent Execution (C1)

The loss mask for selective supervision does not by itself define a safe execution domain: positions without direct supervision may still provide public K/V context, while computations without a direct loss may be predecessors of later supervised predictions on the same TTT path. In the baseline pipeline, the supervision mask restricts output-head and loss computation, but the recurrent TTT unroll still evaluates computations that cannot affect any selected loss (Li et al., 2026). Figure 4 illustrates the distinction. Orange computations directly produce selected losses. Blue computations have no direct loss at their current TTT steps but must execute because they are predecessors of later orange computations on the same TTT path. Gray computations neither produce nor lead to a selected loss and can therefore be skipped. Meanwhile, all TTT-step-1 public K/V remain available because subsequent queries still require the full public context. C1 executes only computations that can reach a current or later selected loss while preserving the full TTT-step-1 public K/V context. Let $\boldsymbol{D}_j$ contain the starting positions that incur a direct selected loss at TTT step j, and let $\boldsymbol{R}_j$ contain those that must execute to reach a selected loss at step j or later:

$$\boldsymbol{D}_j = \{s - j - 1 \mid s \in \boldsymbol{S},\ 1 \leq s - j - 1 \leq L\}, \qquad \boldsymbol{R}_j = \bigcup_{u=j}^{k} \boldsymbol{D}_u. \tag{1}$$

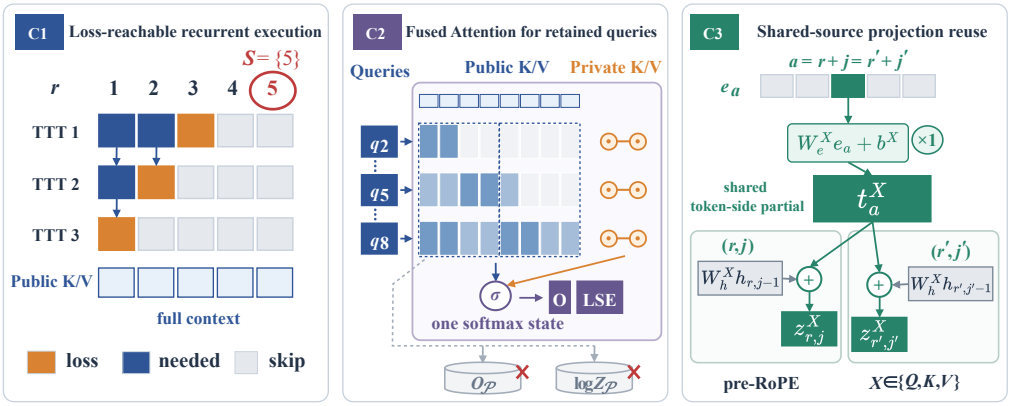

Figure 3: **Overview of Structure-Aware Execution.** In C1, orange, blue, and gray cells denote direct-loss, required-predecessor, and skipped computations respectively. C2 keeps original query positions and avoids storing public-only intermediates, shown as crossed-out buffers; the final output and LSE remain available for backward. C3 reuses t_a^X across computations sharing source position $a = r + j = r' + j'$.

Equation 1 implies $\boldsymbol{D}_j \subseteq \boldsymbol{R}_j$, so C1 executes every computation that directly produces a selected loss at step j. It also implies $\boldsymbol{R}_{j+1} \subseteq \boldsymbol{R}_j$: if C1 retains a computation at step $j+1$, it retains the preceding computation on the same path at step j. Thus, if step u on a TTT path produces a selected loss, C1 also executes steps 1 through $u-1$ on that path. Specifically, for TTT step $j \geq 2$, C1 executes the complete recurrent forward pass only on $\boldsymbol{R}_j$. At TTT step 1, it preserves public K/V preparation over $\boldsymbol{F}$ but skips all other recurrent computations outside $\boldsymbol{R}_1$. C1 therefore removes loss-unreachable work without changing the selected predictions or the training objective. Figure 4 illustrates this execution domain. Appendix A.1 details compact execution and shows why C1 preserves the selected predictions and training objective, including boundary cases.

4.2 Fused Attention for Retained Queries (C2)

C1 retains $M_j = |\boldsymbol{R}_j|$ queries at TTT step j whose original sequence positions may be non-contiguous, while preserving the full public K/V bank. Each query attends to a causal public prefix and, at later TTT steps, a short same-path private history. This requires an attention operator that directly accepts compact queries while preserving their original causal bounds and public context. EAGLE-3 specifies the public-private attention topology, and AngelSpec exploits it through partitioned attention (Li et al., 2025; Liu et al., 2026a; Tencent, 2026). However, the AngelSpec EAGLE-3 path we examined partitions public and private computation while retaining a full-sequence query layout, without integrating C1's query pruning. Its separate public stage also writes public outputs and log-sum-exp values (LSEs), which must be read by a later private merge.

We co-design C2 with C1 to directly process compact queries whose original sequence positions may be non-contiguous, while retaining access to the full public K/V bank. Fusing public and private attention within one forward kernel avoids writing and rereading public-only outputs and LSEs for a separate merge. For a query's public and private key sets $\boldsymbol{\mathcal{P}}$ and $\boldsymbol{\mathcal{H}}$, let $Z_{\boldsymbol{X}}$ denote the sum of exponentiated attention scores and $O_{\boldsymbol{X}}$ the normalized attention output within partition $\boldsymbol{X}$. The joint output is

$$O = \frac{Z_{\boldsymbol{\mathcal{P}}} O_{\boldsymbol{\mathcal{P}}} + Z_{\boldsymbol{\mathcal{H}}} O_{\boldsymbol{\mathcal{H}}}}{Z_{\boldsymbol{\mathcal{P}}} + Z_{\boldsymbol{\mathcal{H}}}}. \tag{2}$$

C2 realizes this identity using a shared online-softmax state within one forward kernel. It saves the final output and LSE for backward, which accumulates gradients from both partitions. Appendix B provides the derivation and implementation details. We also compare C2 with an adapted compact-query attention reference (Tencent, 2026), with C1 and C3 enabled in both configurations. Only the attention implementation differs. Appendix B.1 reports this matched optimizer-step comparison.

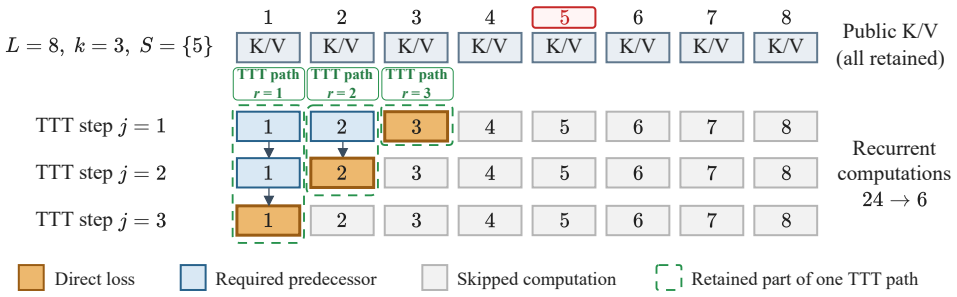

Figure 4: **C1 execution domain for $L = 8$, $k = 3$, and $S = \{5\}$.** Orange cells incur direct loss, blue cells are retained predecessors, gray cells are skipped, and red marks supervision. Full TTT-step-1 public K/V are preserved.

4.3 Shared-Source Projection Reuse (C3)

TTT computations that must be retained can still repeat part of a projection. Computations (r, j) and (r', j') read the same token-side input e_{r+j} when $r + j = r' + j'$. Their hidden-side inputs $h_{r,j-1}$ generally differ because they depend on the states of their respective TTT paths. The existing baselines apply the full Q/K/V projection at every computation, repeating the token-side work even when that input is shared. C3 computes a token-side partial once per required source position and reuses it across TTT paths and TTT steps within the same forward pass. Each computation separately projects its own hidden-side input and adds that result to the shared partial to obtain the complete Q/K/V vectors. For projection $X \in \{Q, K, V\}$, we split the weight into token-side and hidden-side blocks W_e^X and W_h^X, with bias b^X. The pre-RoPE output is

$$z_{r,j}^X = W_e^X e_{r+j} + W_h^X h_{r,j-1} + b^X = t_{r+j}^X + W_h^X h_{r,j-1}, \quad t_a^X = W_e^X e_a + b^X. \tag{3}$$

Here t_a^X is the shared token-side partial at source position a. Each computation adds its own hidden-side contribution before applying the prescribed RoPE to Q/K. To match C1's execution domain, Q partials cover $\boldsymbol{A}_1 = \{r + 1 \mid r \in \boldsymbol{R}_1\}$, whereas K/V partials cover the full domain $\boldsymbol{F}$. Gradients from all uses accumulate into the shared partial. Appendix C details source indexing, gradient equivalence, reuse lifetime, and precision handling.

5 Evaluation

5.1 Experimental Setup

We evaluate training cost on CNN, constructed from CNN/DailyMail (Hermann et al., 2015), and ShareGPT (Chiang et al., 2023). Numerical fidelity and trained-drafter quality are evaluated separately on DRAgent and ShareGPT, using different data splits and training budgets. The training-cost baselines are AngelSpec (Liu et al., 2026a), SpecForge (Li et al., 2026), TorchSpec (TorchSpec team & Mooncake team, 2026), and Speculators (Speculators contributors, 2026).

All reported real-model training, timing, and held-out evaluations use a Qwen3-8B target and an EAGLE-3 drafter on one 48-GB RTX 6000 Ada GPU with BF16 precision. Drafter training uses batch size 1 and TTT length 7. All cost comparisons use offline training with identical input tokens, cached target features, and supervision masks. Reported training time excludes target-feature generation.

For the metrics, E2E time measures the complete training run in seconds. Step wall is the mean complete optimizer-step wall time, including data preparation, forward, backward, optimizer work, and runtime overhead. Forward and Backward are the mean wall times of their corresponding phases per optimizer step. Step and phase times are reported in milliseconds. Allocated peak measures the maximum tensor memory allocated through PyTorch, whereas Reserved peak measures the maximum memory held by its caching allocator, including unused cached blocks. Both are reported in GiB, are not additive, and exclude allocations outside the PyTorch allocator. Table 1 and Panel (b) of Table 3 report arithmetic means over ten independently launched complete runs per framework

Table 1: **External training-cost comparison on CNN (Hermann et al., 2015) and ShareGPT (Chiang et al., 2023).**

Metric	Ours	AngelSpec	SpecForge	TorchSpec	Speculators
CNN					
E2E time (s)	935.85	3184.45	3202.31	3251.65	4058.96
Forward (ms)	200.97	545.70	566.73	587.50	838.38
Backward (ms)	180.10	881.59	876.37	887.46	998.86
Allocated peak (GiB)	21.34	32.12	33.21	34.57	46.82
Reserved peak (GiB)	21.68	33.01	33.56	34.91	47.03
ShareGPT					
E2E time (s)	5,591.56	10,337.79	10,416.21	10,860.53	12,014.84
Forward (ms)	109.39	202.96	201.35	210.66	230.47
Backward (ms)	101.64	244.94	249.78	250.83	295.40
Allocated peak (GiB)	14.51	20.45	21.53	21.19	28.97
Reserved peak (GiB)	14.83	20.69	21.74	21.33	29.14

or configuration. Appendix E provides the detailed data splits, training budgets, offline data preparation, and run-level aggregation procedures.

5.2 Matched Training Efficiency and Peak Memory

We benchmark our method against four external training frameworks on CNN and ShareGPT. Under the common training setup in Section 5.1, we compare training time and peak PyTorch memory. Table 1 shows that our method achieves the lowest end-to-end training time and the lowest allocated and reserved memory peaks on both workloads. On CNN, our method achieves a $3.40\times$–$4.34\times$ end-to-end training speedup over the four external baselines and reduces PyTorch allocated peak memory by 33.56%–54.42%. On ShareGPT, our method achieves a $1.85\times$–$2.15\times$ end-to-end training speedup and reduces PyTorch allocated peak memory by 29.05%–49.91%, relative to the same four baselines. The larger speedups on CNN are consistent with its sparser direct supervision, which provides more opportunities to eliminate loss-unreachable recurrent work. Sections 5.4 and 5.5 further examine this relationship through component ablations and controlled changes in supervision density and layout.

5.3 Numerical Fidelity and Trained-Drafter Quality

We first compare our method with SpecForge on DRAgent and ShareGPT using the same model parameters, optimizer state, and inputs to assess numerical agreement. We then evaluate the trained checkpoints on held-out inputs to measure draft acceptance and inference throughput. In Panel (a) of Table 2, Output rel.-L_2 measures error in the selected draft outputs; Loss abs. gap is the absolute scalar-loss difference; Gradient rel.-L_2 uses concatenated trainable-parameter gradients; Worst tensor rel.-L_2 reports the largest per-parameter-tensor error; and Update rel.-L_2 compares optimizer displacement vectors $\Delta\theta$. Relative errors use the corresponding SpecForge tensor norm as the denominator. Appendix E.3 details the paired comparisons and checkpoint evaluation.

Across ten paired training seeds, the mean all-gradient relative-L_2 error is at most 0.62% on both datasets, with no missing or non-finite gradients. Panel (b) shows comparable held-out acceptance and throughput: accepted drafts per verification differ from SpecForge by at most 1.06%, relative to the SpecForge value. Both trained methods outperform Vanilla, the unadapted EAGLE-3 checkpoint, indicating that our execution changes preserve the benefits of drafter training. We also compare the paired training-loss curves over the complete training runs, with Figure 5 and the detailed loss-gap analysis provided in Appendix E.4.

5.4 Component Ablations

We use component ablations on CNN and ShareGPT to measure savings from skipping recurrent work, sharing token-side projections, and fusing attention for retained queries. Table 3 reports structure statistics, full-run timing, and peak memory for these execution choices. C1 produces the largest single reduction. Panel (a) shows that CNN has much sparser direct supervision than

Table 2: **Numerical fidelity and trained-drafter quality.** Panel (a) reports matched numerical fidelity; Panel (b) reports held-out acceptance and throughput.

(a) Matched numerical fidelity

Dataset	Output rel.-L_2 (%)	Loss abs. gap	Gradient rel.-L_2 (%)	Worst tensor rel.-L_2 (%)	Update rel.-L_2 (%)
DRAgent	1.24	3.02×10^{-5}	0.62	Q proj., 1.02	0.94
ShareGPT	1.07	1.20×10^{-3}	0.58	Q proj., 0.83	0.88

(b) Acceptance and throughput

Metric	Vanilla	SpecForge	Our method
DRAgent (200 Test prefixes)			
Advanced tokens / verification	1.74	2.13	2.14
Accepted drafts / verification	0.74	1.13	1.14
Throughput (tokens/s)	43.77	50.89	51.23
ShareGPT (500 Test conversations)			
Advanced tokens / verification	3.47	3.83	3.86
Accepted drafts / verification	2.47	2.83	2.86
Throughput (tokens/s)	117.80	128.71	129.30

Table 3: **Full-run component ablation, execution structure, and peak memory.** Panel (a) uses the complete cost workloads. Public K/V coverage divides covered source positions by all valid source positions. Panel (b) uses SpecForge as the baseline. C3 build is the scoped cost of constructing the reusable source tables.

(a) Executed and reused work

Dataset	Supervision	Public K/V coverage	Q reuse	K/V reuse
CNN	6.75%	100.00%	6.96×	1.41×
ShareGPT	56.55%	100.00%	6.93×	4.44×

(b) Full-run component ablation

Dataset	Configuration	Step wall (ms)	Forward (ms)	Backward (ms)	C3 build (ms)	Allocated peak (GiB)	Reserved peak (GiB)
CNN	SpecForge	1501.78	566.73	876.37	0.00	33.21	33.56
CNN	C1	571.18	230.94	280.52	0.00	25.83	26.21
CNN	C1 + C2	511.34	201.49	250.23	0.00	25.76	26.02
CNN	C1 + C3	499.52	228.91	210.16	3.07	21.64	21.89
CNN	Full	**441.55**	**200.97**	**180.10**	2.56	**21.34**	**21.68**
ShareGPT	SpecForge	510.42	201.35	249.78	0.00	21.53	21.74
ShareGPT	C1	353.73	127.88	165.45	0.00	17.08	17.22
ShareGPT	C1 + C2	313.51	110.58	143.47	0.00	17.02	17.26
ShareGPT	C1 + C3	304.84	126.23	118.25	2.56	14.67	14.96
ShareGPT	Full	**270.37**	**109.39**	**101.64**	2.33	**14.51**	**14.83**

ShareGPT. C1 therefore removes more loss-unreachable recurrent work and yields a larger step-wall improvement on CNN, while still benefiting ShareGPT. Both workloads preserve complete public K/V coverage.

Relative to C1, adding C2 reduces step wall and both forward and backward time on both workloads, while changing allocated peak only marginally. This pattern is expected because C2 changes how the retained asymmetric attention domain is executed without changing C1's retained-query set or complete public K/V coverage. With C1 and C3 enabled, we also compare C2 with an adapted compact-query attention reference in Appendix B.1.

C3 reuses token-side projections across TTT computations that read the same source position. Table 3 reports the net effect of enabling C3, including the cost of building the shared tables. Relative to C1, C1+C3 has similar forward time but lower backward time, step wall, and allocator peaks on both workloads. The full method yields the lowest step wall and allocator peaks. Appendix F defines the execution and measurement ratios.

Table 4: **Sensitivity to supervision density and layout.** Synthetic 12,288-token inputs, TTT length 7, and 2,000 optimizer steps. Speedup compares mean step wall (SpecForge (Li et al., 2026)/Ours); memory columns give our method's peak reductions relative to SpecForge. Random rows average 10 seeds.

Config. density (%)	Layout	Supervised positions	Step-wall speedup (×)	Allocated peak reduction (%)	Reserved peak reduction (%)
25	Tail	3,072	2.80	54.80	47.50
25	Uniform	3,072	2.08	38.60	43.20
25	Random	3,072	2.34	43.33	43.90
50	Tail	6,144	2.17	39.70	49.20
50	Uniform	6,144	1.58	26.50	29.10
50	Random	6,144	1.76	28.72	31.94
75	Tail	9,216	1.55	24.40	28.40
75	Uniform	9,216	1.24	16.80	18.14
75	Random	9,216	1.35	18.18	19.88
100	Full	12,287	1.18	11.92	13.26

5.5 Supervision-Density and Layout Sensitivity

We vary supervision density and placement on synthetic training inputs to test how loss dependencies affect optimizer-step savings at a fixed sequence length and TTT length. Table 4 shows that the speed and allocated-memory gains generally diminish as supervision becomes denser. Denser supervision keeps more starting positions and TTT computations live, leaving less opportunity for C1 to remove work. At full supervision, supervision-driven sparsity largely disappears, while the complete method continues to reduce step wall and allocator-visible memory relative to SpecForge.

Layout changes the overlap among these loss-reachable sets even when the number of supervised positions is fixed. With Tail supervision, selected positions are contiguous, so the shifted direct-loss sets $\boldsymbol{D}_j$ overlap substantially across TTT steps. Their unions $\boldsymbol{R}_j$ remain relatively compact. Uniform supervision spreads selected positions across the sequence; shifting these positions for different TTT steps produces less overlap and covers more distinct TTT-path starting positions. C1 must consequently retain a larger and more fragmented execution domain, which reduces the savings available to C1 and increases the work presented to C2 and C3. Random layouts fall between these cases on average because some selected positions form local clusters. This explains why Tail consistently has the highest speedup, whereas Uniform is lowest at the non-full densities. At full supervision, all loss-eligible positions are selected, so the three layouts coincide and the Full row does not compare placement strategies. Reserved-memory reductions are less monotonic because they also reflect allocator reservation behavior and should therefore be interpreted separately from allocated tensor memory. Overall, the results show that supervision placement matters in addition to density before the full-supervision endpoint. Appendix I reports additional synthetic-data step-wall speedups across sequence lengths and TTT lengths.

6 Limitations

Our current implementation targets EAGLE-3's same-path training topology. Future work can explore more general draft structures and evaluate the resulting methods on broader training and deployment workloads.

7 Conclusion

Structure-Aware Execution uses the TTT unroll to identify recurrent work that can be skipped and token-side projections that can be shared. It preserves required same-path states and full public K/V, fuses attention for retained queries, and reuses projections by source position. Our evaluation shows lower training cost with comparable trained-drafter quality and explains how supervision density and layout affect the savings.

AI USE STATEMENT

We used generative AI tools to assist with manuscript preparation, including drafting, language polishing, and figure preparation. The authors take responsibility for the final content of this paper.

REFERENCES

Zihao An, Huajun Bai, Ziqiong Liu, Dong Li, and Emad Barsoum. PARD: Accelerating LLM Inference with Low-Cost PARallel Draft Model Adaptation. arXiv preprint arXiv:2504.18583v4, 2025. URL https://arxiv.org/abs/2504.18583.

Jeremy Appleyard, Tomáš Kočiský, and Phil Blunsom. Optimizing Performance of Recurrent Neural Networks on GPUs. arXiv preprint arXiv:1604.01946v1, 2016. URL https://arxiv.org/html/1604.01946v1.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016. URL https://arxiv.org/abs/1604.06174.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. LMSYS Org blog, 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

João Coelho, Jingjie Ning, Jingyuan He, Kangrui Mao, Abhijay Paladugu, Pranav Setlur, Jiahe Jin, Jamie Callan, João Magalhães, Bruno Martins, and Chenyan Xiong. DeepResearchGym: A Free, Transparent, and Reproducible Evaluation Sandbox for Deep Research. *arXiv preprint arXiv:2505.19253*, 2025. URL https://arxiv.org/abs/2505.19253.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *arXiv preprint arXiv:2205.14135*, 2022. URL https://arxiv.org/abs/2205.14135.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. DeepResearch Bench: A Comprehensive Benchmark for Deep Research Agents. *arXiv preprint arXiv:2506.11763*, 2025. URL https://arxiv.org/abs/2506.11763.

Karl Moritz Hermann, Tomáš Kočiský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. In *Advances in Neural Information Processing Systems*, volume 28, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/afdec7005cc9f14302cd0474fd0f3c96-Abstract.html.

Fenglu Hong, Ravi Raju, Jonathan Lingjie Li, Bo Li, Urmish Thakker, Avinash Ravichandran, Swayambhoo Jain, and Changran Hu. Training Domain Draft Models for Speculative Decoding: Best Practices and Insights. arXiv preprint arXiv:2503.07807v2, 2025. URL https://arxiv.org/html/2503.07807v2.

Mude Hui, Xin Huang, Jaime Campos Salas, Yue Sun, Nathan Pemberton, Xiang Song, Ashish Khetan, and George Karypis. P-EAGLE: Parallel-Drafting EAGLE with Scalable Training. arXiv preprint arXiv:2602.01469v1, 2026. URL https://arxiv.org/html/2602.01469v1.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-Throughput LLM Inference with Shared Prefixes. arXiv preprint arXiv:2402.05099v2, 2024. URL https://arxiv.org/html/2402.05099v2.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286. PMLR, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Shenggui Li, Chao Wang, Yikai Zhu, Yubo Wang, Fan Yin, Shuai Shi, Yefei Chen, Xiaomin Dong, Qiaoling Chen, Jin Pan, Ji Li, Laixin Xie, Yineng Zhang, Lei Yu, Yonggang Wen, Ivor Tsang, and Tianwei Zhang. SpecForge: A Flexible and Efficient Open-Source Training Framework for Speculative Decoding. *arXiv preprint arXiv:2603.18567*, 2026. URL https://arxiv.org/html/2603.18567v1.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test. *arXiv preprint arXiv:2503.01840*, 2025. URL https://arxiv.org/html/2503.01840v3.

Hong Liu, Rui Cen, Junhan Shi, Guangshuo Qin, Jiebin Zhang, Tianyu Liu, Runzhi Fan, Guoliang Zhao, Ruobing Xie, Kai Zhang, Song Liu, Guanghua Yu, and Jianchen Zhu. AngelSpec: Towards Real-World High Performance Inference with Speculative Decoding. arXiv preprint arXiv:2607.25852v1, 2026a. URL https://arxiv.org/html/2607.25852v1.

Hongyi Liu, Jiaji Huang, Zhen Jia, Youngsuk Park, and Yu-Xiang Wang. Not-a-bandit: Provably no-regret drafter selection in speculative decoding for LLMs. In *International Conference on Learning Representations*, 2026b. URL https://arxiv.org/abs/2510.20064v2. arXiv:2510.20064v2.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Online Speculative Decoding. arXiv preprint arXiv:2310.07177v4, 2023. URL https://arxiv.org/abs/2310.07177.

NVIDIA. Model Optimizer: Speculative Decoding, 2026. URL https://github.com/NVIDIA/Model-Optimizer/blob/f335459dc053895bd7b0a77ecaa47e87c2080ab3/examples/speculative_decoding/README.md. Repository documentation at commit f335459dc053895bd7b0a77ecaa47e87c2080ab3 (2026-06-27); accessed 2026-09-13.

Corby Rosset, Ho-Lam Chung, Guanghui Qin, Ethan C. Chau, Zhuo Feng, Ahmed Awadallah, Jennifer Neville, and Nikhil Rao. Researchy Questions: A Dataset of Multi-Perspective, Decompositional Questions for LLM Web Agents. *arXiv preprint arXiv:2402.17896*, 2024. URL https://arxiv.org/abs/2402.17896.

Alexander Samarin, Sergei Krutikov, Anton Shevtsov, Sergei Skvortsov, Filipp Fisin, and Alexander Golubev. LK Losses: Direct Acceptance Rate Optimization for Speculative Decoding. arXiv preprint arXiv:2602.23881v2, 2026. URL https://arxiv.org/abs/2602.23881.

Speculators contributors. Speculators, 2026. URL https://github.com/vllm-project/speculators/tree/35081a47fa5881504c35ad9a7be25af742b91e52. commit 35081a47fa5881504c35ad9a7be25af742b91e52; accessed 2026-09-13.

Tencent. AngelSpec: EAGLE-3 Training Attention Implementation, 2026. URL https://github.com/Tencent/AngelSpec/blob/d3412bed231025ea85d35dbf5e0af44f1ac62a5b/angelspec/models/draft/llama3_eagle.py. commit d3412bed231025ea85d35dbf5e0af44f1ac62a5b; accessed 2026-09-13.

TorchSpec team and Mooncake team. TorchSpec: Speculative Decoding Training at Scale, 2026. URL https://pytorch.org/blog/torchspec-speculative-decoding-training-at-scale/. Official technical article, 2026-03-19; accessed 2026-09-13.

Junxiong Wang, Fengxiang Bie, Jisen Li, Zhongzhu Zhou, Zelei Shao, Yubo Wang, Yinghui Liu, Qingyang Wu, Avner May, Sri Yanamandra, Ce Zhang, Tri Dao, Percy Liang, Ben Athiwaratkun, Shuaiwen Leon Song, Chenfeng Xu, and Xiaoxia Wu. When RL Meets Adaptive Speculative Training: A Unified Training-Serving System. arXiv preprint arXiv:2602.06932v5, 2026a. URL https://arxiv.org/abs/2602.06932.

Zili Wang, Zhaopeng Qiu, Yuekai Zhang, Shuang Yu, and Junjie Lai. Online Draft Co-Training for Speculative Decoding in Large-Scale, Long-Context RL Post-Training. arXiv preprint arXiv:2609.07108v1, 2026b. URL https://arxiv.org/abs/2609.07108.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. arXiv preprint arXiv:2501.01005v1, 2025. URL https://arxiv.org/html/2501.01005v1.

Ofir Zafrir, Igor Margulis, Dorin Shteyman, Shira Guskin, and Guy Boudoukh. FastDraft: How to Train Your Draft. arXiv preprint arXiv:2411.11055v3, 2024. URL https://arxiv.org/abs/2411.11055.

Lefan Zhang, Xiaodan Wang, Yanhua Huang, and Ruiwen Xu. Learning Harmonized Representations for Speculative Sampling. arXiv preprint arXiv:2408.15766v3, 2024. URL https://arxiv.org/abs/2408.15766.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. DistillSpec: Improving Speculative Decoding via Knowledge Distillation. arXiv preprint arXiv:2310.08461v2, 2023. URL https://arxiv.org/abs/2310.08461.

A NOTATION AND WHY C1 PRESERVES THE LOSS

EAGLE-3 runs its drafter for several recurrent steps during training. This procedure is called training-time test (TTT). A TTT path is the sequence of recurrent computations that share a starting position r, as defined in Section 3. Public keys and values (K/V) are built at TTT step 1 and provide context across paths. Private K/V come from earlier steps on the same path. C1 keeps the full set of positions $\boldsymbol{F}$ for public K/V, but runs the remaining recurrent work only at starting positions in $\boldsymbol{R}_j$. Table 5 lists the symbols used below.

A.1 WHAT C1 KEEPS AND WHY THE RESULT IS UNCHANGED

At TTT step 1, C1 builds public K/V for all positions in $\boldsymbol{F}$ because retained queries may read them. It runs the other recurrent computations only at starting positions in $\boldsymbol{R}_1$. At each later step j, it runs the recurrent drafter only at starting positions in $\boldsymbol{R}_j$. The set $\boldsymbol{D}_j$ contains the starting positions that produce a direct loss at step j. Only these positions pass through output normalization, the drafter's vocabulary projection (the output head), and the loss calculation.

Figure 4 illustrates why C1 keeps some computations that have no direct loss. The example has a sequence of $L = 8$ positions, $k = 3$ TTT steps, and direct supervision only at token position 5 ($\boldsymbol{S} = \{5\}$). The grid contains one candidate computation (r, j) for each starting position and TTT step, giving $8 \cdot 3 = 24$ cells. The red mark identifies the supervised token; orange cells directly predict it, blue cells have no loss at their current TTT step but lead to a later selected loss on the same TTT path, and gray cells are skipped because they cannot affect a selected loss. The TTT-step-1 public K/V still span all eight positions. A cell with $r + j + 1 \notin \boldsymbol{F}$ is masked at the sequence boundary and adds no loss.

C1 stores the states at each TTT step in a compact list ordered by starting position. For an original starting position $r \in \boldsymbol{R}_j$, $\pi_j(r)$ gives its index in that list. At TTT step $j \geq 2$, C1 finds the previous state $h_{r,j-1}$ at $\pi_{j-1}(r)$, so the computation stays on the same TTT path. It likewise finds an earlier private K/V entry using $\pi_e(r)$ for its TTT step e. For $r \in \boldsymbol{D}_j$, the selected loss reads $h_{r,j}$ at $\pi_j(r)$ but still uses the teacher distribution for the original token position $s = r + j + 1$. Public K/V keep their full-sequence positions in $\boldsymbol{F}$.

To show why C1 keeps the same supervised predictions, compare it with the baseline using the same model parameters, target features, token-side inputs, original positions, attention masks, and values from any random operations. Each query can read the same public positions allowed by its original causal mask. From TTT step 2 onward, it can also read private K/V from earlier steps on its own path.

On the path starting at $r = 1$ in Figure 4, $(1, 3)$ produces a selected loss and requires $(1, 1)$ and $(1, 2)$ as preceding computations. In general, a selected loss at TTT step u on path r means $r \in \boldsymbol{D}_u$. Since $\boldsymbol{R}_j = \bigcup_{v=j}^{k} \boldsymbol{D}_v$, that same r belongs to every $\boldsymbol{R}_j$ for $1 \leq j \leq u$. C1 therefore keeps every preceding computation on that TTT path.

The retained computations also produce the same states as in the baseline. At TTT step 1, a retained query uses the same inputs, original position, attention mask, and full public K/V, so its state is unchanged. Suppose the retained states through TTT step $j - 1$ are unchanged. Because $\boldsymbol{R}_j \subseteq \boldsymbol{R}_{j-1}$, C1 can read the same previous state for each $r \in \boldsymbol{R}_j$. Its query then reads the same public K/V and earlier same-path private K/V. Thus its state and draft distribution at TTT step j are unchanged as well.

Every selected loss uses one of these unchanged outputs because $\boldsymbol{D}_j \subseteq \boldsymbol{R}_j$. C1 also keeps the same teacher probabilities, vocabulary mapping, supervision mask, weight for each TTT step, and denominator used to normalize the loss. If calculations have no floating-point rounding, we call this exact arithmetic. Under this assumption, the total loss and its parameter gradients are unchanged wherever the derivatives exist. An unsupervised position can still receive gradients through its public K/V. Keeping all public K/V preserves those gradient contributions.

This argument requires private attention to stay within each TTT path, all public K/V to remain available, and the loss denominator to stay unchanged. If a sample has no selected positions, the implementation uses the existing full execution without C1 pruning. This is the fallback for that case.

Table 5: **Notation used throughout the paper.** C1 packs retained computations into smaller tensors, but each computation keeps its original sequence position.

Symbol	Meaning
L	Sequence length
$\boldsymbol{F} = \{1, \ldots, L\}$	All sequence positions
$\boldsymbol{S} \subseteq \boldsymbol{F}$	Token positions selected as training targets
k	Number of recurrent steps in training-time test (TTT)
$j \in \{1, \ldots, k\}$	TTT step j along a TTT path
$r \in \boldsymbol{F}$	Starting position of a TTT path
(r, j)	Computation at TTT step j on the path starting at r; predicts the token at position $r + j + 1$ when that position lies in $\boldsymbol{F}$
x_a	Token at position a in the training sequence; read by the drafter
$\tilde{x}_a$	Token predicted by the target LLM at position a
$(f_a)_{a \in \boldsymbol{F}}$	Target-derived features $(f_1, \ldots, f_L)$; f_a comes from selected target-model hidden layers at position a
e_a	Token-side input from x_a after embedding and input preprocessing
$h_{r,0}$	Initial hidden-side state on path r, derived from f_r after preprocessing
$h_{r,j}$	State produced by (r, j) and used as hidden-side input at TTT step $j + 1$ after preprocessing
$\hat{p}_{r,j}$	Token probabilities predicted by the drafter at (r, j)
$\tilde{p}_s$	Target-model token probabilities used to supervise position $s = r + j + 1$
$\boldsymbol{D}_j = \{r \in \boldsymbol{F} \mid r + j + 1 \in \boldsymbol{S}\}$	Starting positions with a direct selected loss at TTT step j
$\boldsymbol{R}_j = \bigcup_{u=j}^{k} \boldsymbol{D}_u$	Starting positions kept at TTT step j to reach a selected loss at step j or later
$\boldsymbol{A}_1 = \{r + 1 \mid r \in \boldsymbol{R}_1\}$	Token-side source positions used by retained TTT-step-1 computations
$\pi_j(r)$	Index of starting position r in the compact list $\boldsymbol{R}_j$
M_j	Number of retained queries at TTT step j: $\lvert\boldsymbol{R}_j\rvert$
N	Public-K/V length (L before padding)
d	Number of same-path private K/V entries per retained query
$\boldsymbol{\mathcal{P}}$	Public keys visible to a retained query
$\boldsymbol{\mathcal{H}}$	Same-path private keys visible to a retained query
q, k_i, v_i	Query, key, and value vectors
d_h	Attention-head dimension
$s_i = q^\top k_i / \sqrt{d_h}$	Attention score for visible key i
$Z_{\boldsymbol{X}}$	Sum of exponentiated scores over key set $\boldsymbol{X} \in \{\boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{H}}\}$
$N_{\boldsymbol{X}}$	Sum of value vectors weighted by exponentiated scores over key set $\boldsymbol{X}$
$O_{\boldsymbol{X}}$	Attention output using only key set $\boldsymbol{X}$: $N_{\boldsymbol{X}}/Z_{\boldsymbol{X}}$
$X \in \{Q, K, V\}$	Q, K, or V projection
W_e^X	Token-side weight for projection X
W_h^X	Hidden-side weight for projection X
b^X	Bias for projection X
$z_{r,j}^X$	Full output of projection X at (r, j), before rotary position embedding (RoPE)
t_a^X	Token-side part of projection X at source position a, reused across computations
$\boldsymbol{\mathcal{U}}_a^X$	Executed TTT computations with $r + j = a$ that reuse token-side partial t_a^X
$\mathcal{L}$	Scalar training loss
$\overline{x}$	Loss gradient with respect to x
$u_{r,j}^X$	Loss gradient with respect to projection output $z_{r,j}^X$

Any placeholder rows required by the implementation are masked and add no loss. The sample still follows the usual optimizer schedule: even with zero loss, skipping an update can change the optimizer state. Together, these rules give an execution domain that preserves the selected predictions, loss, and gradients in exact arithmetic.

B Fused Attention for Retained Queries (C2)

For one retained query, let $\boldsymbol{\mathcal{P}}$ contain its visible public keys and $\boldsymbol{\mathcal{H}}$ its same-path private keys. These two sets are the public and private attention partitions. The score for key i is $s_i = q^\top k_i/\sqrt{d_h}$, where d_h is the number of elements in one attention-head vector. For either key set $\boldsymbol{X}$, let $Z_{\boldsymbol{X}} = \sum_{i\in\boldsymbol{X}} \exp(s_i)$ be the sum used to normalize attention weights. Let $N_{\boldsymbol{X}} = \sum_{i\in\boldsymbol{X}} \exp(s_i)v_i$ be the corresponding weighted sum of value vectors. For a nonempty set, its attention output is $O_{\boldsymbol{X}} = N_{\boldsymbol{X}}/Z_{\boldsymbol{X}}$. No key entry belongs to both partitions, so their sums can be added: $Z = Z_{\boldsymbol{\mathcal{P}}} + Z_{\boldsymbol{\mathcal{H}}}$ and $N = N_{\boldsymbol{\mathcal{P}}} + N_{\boldsymbol{\mathcal{H}}}$. The full output $O = N/Z$ then gives Equation 2. At TTT step 1, there is no private history: $\boldsymbol{\mathcal{H}} = \varnothing$ and $Z_{\boldsymbol{\mathcal{H}}} = N_{\boldsymbol{\mathcal{H}}} = 0$. The output is therefore just $O_{\boldsymbol{\mathcal{P}}}$.

C2 handles both key sets in one forward GPU kernel. It processes public K/V in small matrix blocks, called tiles. For private K/V, it directly computes the dot product for each existing same-path query-key pair. The number of queries can be smaller than the public-K/V length. Their original positions may also have gaps, so the kernel uses each query's original position to decide which public keys it can read. Before the kernel runs, the caller gathers private K/V by TTT path so that each query receives the correct history. The kernel does not need to search for matching paths.

C2 updates three running quantities as it reads public tiles and private entries: the largest score seen so far, a rescaled sum of exponentiated scores, and a rescaled weighted sum of values. Together, these quantities form the online-softmax state. They allow one shared normalization across both key sets without storing all attention scores. The log-sum-exp (LSE) is the logarithm of the normalization sum, $\log Z$. C2 does not write a separate public-only output or public-only LSE for a later kernel to merge. It saves the final attention output and LSE over both sets, which are needed to compute gradients during the backward pass.

Let $M_j = |\boldsymbol{R}_j|$ be the number of retained queries, L the public-K/V length, and d the number of private entries per query. Before causal masking, public attention requires up to $\mathcal{O}(M_j L d_h)$ arithmetic work per head. Private attention requires $\mathcal{O}(M_j d d_h)$. The public length L can be large, while d is at most the number of earlier TTT steps. This difference is why C2 uses matrix tiles for public attention and individual row-wise dot products for private attention.

One recorded training example shows the size difference. At TTT step 7, the sequence has $L = 10{,}546$ positions, $M_7 = 818$ retained queries, and $d = 6$ private keys per query. Public attention has $818 \times 10{,}546 = 8{,}626{,}628$ possible query-key pairs per head before causal masking. Private attention has only $818 \times 6 = 4{,}908$ pairs, about 1.76×10^3 times fewer. Causal masking removes some public pairs, but the remaining public work still benefits from matrix tiles. Private attention needs only a few dot products per query.

The backward pass uses the final output and LSE saved in 32-bit floating-point format (FP32). It sums the public and private contributions to the query gradient and computes the gradients of their keys and values using the same normalization. The backward pass uses multiple kernels. For grouped-query attention, where several query heads share one K/V head, it also sums those heads' contributions to the shared K/V gradients.

To save memory, activation checkpointing discards some intermediate tensors and recomputes them during backward. This differs from saving model weights to a checkpoint file. We compile the attention function used by activation checkpointing together with its tensor-layout preparation, saved-state handling, and output slicing. Their costs, including recomputation, are included in the measurements. The compiled code is specialized for the head dimensions used in the experiments. When separate arguments refer to shared data, the implementation gives each argument its own tensor view, which refers to the same underlying storage. This keeps the gradient contribution from every use, as required by C3. CUDA Graph execution, which records and replays GPU launches, is disabled.

B.1 Comparing Attention Implementations Within a Training Step

We compare C2 with a reference adapted from AngelSpec (Tencent, 2026) to accept C1's retained queries in Table 6. This is the adapted compact-query attention reference named in the main text. It computes public attention first, then merges same-path private attention using the same softmax normalization. Both configurations enable C1 and C3. They use the same attention inputs, original positions, model parameters, optimizer state, and sample order. Only the attention implementation changes. The C2 column corresponds to the Full (C1+C2+C3) row of Table 3; the adapted-reference column is separate from the C1+C3 row of Table 3.

Private gather means collecting earlier private K/V in the order needed by the retained queries. Both implementations use the same gather operation. Its timing is already included in the larger measured operations, so the table rows overlap and must not be added to obtain the complete step time. Step wall is the elapsed time for a complete optimizer step. Speedup is the reference time divided by the C2 time.

Table 6: **Attention comparison with all other settings fixed.** Private gather measures the shared work that aligns private K/V with retained queries over TTT steps 2–7.

Dataset	Metric	Adapted ref. (ms)	C2 (ms)	Speedup (×)
CNN	Step wall	466.98	441.55	1.06
	Forward	215.48	200.97	1.07
	Backward	191.46	180.10	1.06
	Private gather (shared)	1.55	1.55	1.00 / shared
ShareGPT	Step wall	285.72	270.37	1.06
	Forward	116.63	109.39	1.07
	Backward	109.51	101.64	1.07
	Private gather (shared)	2.10	2.10	1.00 / shared

Table 6 shows that C2 reduces forward, backward, and complete optimizer-step time on both datasets. The shared private-gather cost stays the same. These results support the benefit of C2's implementation (Section 4.2), which uses one online-softmax state and avoids writing and rereading public-only outputs and LSE values for a separate merge.

C Shared-Source Projection Gradients and Reuse Limits (C3)

For each query, key, or value projection $X \in \{Q, K, V\}$, the original implementation joins the token-side and hidden-side inputs into one vector before multiplying by W^X. We split the columns of this weight matrix into two blocks: $W^X = [W_e^X \; W_h^X]$. The first block multiplies the token-side input e_{r+j}; the second multiplies the hidden-side input $h_{r,j-1}$ after the same preprocessing as in the baseline. This does not change any parameter. By linearity, the result equals the sum of the two products, as shown in Equation 3.

The token-side partial is the part of the projection that depends only on the token-side input: $t_a^X = W_e^X e_a + b^X$. We store these partials in a source table indexed by source position a. Each computation reads t_{r+j}^X and adds its own hidden-side product $W_h^X h_{r,j-1}$. The bias is already in the shared partial, so it is added only once. Rotary position embedding (RoPE) applies a position-dependent rotation to Q and K. We apply it after adding the two parts, using the original position indices.

Sharing is based on the original source position within one sample. If $r + j = r' + j' = a$, the two computations read the same token-side vector e_a under the training setup studied here. They can therefore reuse t_a^X. A matching token ID at another position is not enough to enable sharing, and we do not share entries across samples. Each computation still projects its own hidden-side input.

For projection $X \in \{Q, K, V\}$, the output before RoPE is $z_{r,j}^X = t_{r+j}^X + W_h^X h_{r,j-1}$, where $t_a^X = W_e^X e_a + b^X$. The following set lists all executed computations that use the partial at source position

a:

$$\boldsymbol{\mathcal{U}}_a^X = \{(r,j) \mid \text{projection } X \text{ is executed for } (r,j) \text{ and } r+j=a\}\,. \tag{4}$$

Let $u_{r,j}^X = \partial\mathcal{L}/\partial z_{r,j}^X$ be the loss gradient at the projection output before RoPE. It includes the effect of RoPE and all later operations. An overbar denotes a loss gradient. The superscript (X) on $\overline{e}_a^{(X)}$ and $\overline{h}_{r,j-1}^{(X)}$ identifies the contribution from projection X. Every use of a shared partial contributes to its gradient, so these contributions must be added:

$$\bar{t}_a^X = \sum_{(r,j)\in\boldsymbol{\mathcal{U}}_a^X} u_{r,j}^X, \qquad \overline{W}_e^X = \sum_a \bar{t}_a^X e_a^\top, \qquad \bar{b}^X = \sum_a \bar{t}_a^X, \tag{5}$$

$$\overline{e}_a^{(X)} = (W_e^X)^\top \bar{t}_a^X, \qquad \overline{W}_h^X = \sum_a \sum_{(r,j)\in\boldsymbol{\mathcal{U}}_a^X} u_{r,j}^X h_{r,j-1}^\top, \qquad \overline{h}_{r,j-1}^{(X)} = (W_h^X)^\top u_{r,j}^X. \tag{6}$$

The first equation adds the gradient from every use of t_a^X. The remaining equations pass that sum to the token-side weights, bias, and input, while keeping the hidden-side gradients separate for each computation. Summing the input gradients over $X \in \{Q,K,V\}$ gives the total gradients. Expanding $\bar{t}_a^X$ gives the same result as projecting the token input separately at every computation. If e_a comes from a trainable normalization layer, that layer also receives the summed gradient. Q, K, and V may be used by different sets of computations, so their reuse sets must be tracked separately. When a gather reads one table entry multiple times, its backward pass must add those contributions rather than overwrite them.

The Q and K/V tables need different source positions. To see why all retained Q computations can use the TTT-step-1 source set $\boldsymbol{A}_1$, take any retained computation (r,j) with $r \in \boldsymbol{R}_j$. Some step $u \geq j$ on this path predicts a supervised position $s \in \boldsymbol{S}$, so $r = s-u-1$. The current token-side source is $r+j = s-(u-j+1)$. Now define $r' = r+j-1 = s-(u-j+2)$. This starting position predicts s at step $u-j+1$, because $r' + (u-j+1) + 1 = s$. Since $1 \leq u-j+1 \leq k$, we have $r' \in \boldsymbol{D}_{u-j+1} \subseteq \boldsymbol{R}_1$. Thus $r+j = r'+1 \in \boldsymbol{A}_1$.

K/V also need all source positions in $\boldsymbol{F}$ to build the full public context at TTT step 1. The implementation stores the retained source positions first in the K/V table, followed by the other required positions. A map from each original source position to its table index locates the correct entry. Q uses a separate map over retained source positions. Placeholder rows, shifted positions outside the sequence, and fallback cases still follow the baseline's masks and index checks. The proof above concerns valid retained computations.

A source table can be reused within one forward TTT unroll, meaning one traversal of its recurrent steps, because the token-side inputs and projection weights stay fixed during that traversal. The hidden-side inputs can change. For example, let two computations have scalar hidden inputs zero and one, with $W_h = 1$. Reusing the complete projection would make the second output wrong by one. We therefore share only the token-side partial.

Sharing a partial after RoPE can also be wrong when the two computations use different rotation positions. C3 instead shares partials before RoPE and applies the required rotation after combining the token-side and hidden-side parts. After an optimizer update, the table must be rebuilt because W_e may have changed. Sharing also requires the same token-side vector, not just the same source index. An alternative training setup that feeds the drafter's own generated tokens back into later steps could violate this requirement.

For a simple gradient example, let $e_a = 1$ and let two computations each contribute a gradient of one at their projection output. The gradient of W_e must then be two. Disconnecting the shared table from automatic differentiation would give zero; counting only one use would give one. Reading the wrong source entry, adding the bias twice, or reusing a table after its weights change can also alter the result. These examples explain why we check source indices, forward outputs, and gradients separately.

We check the projection identity with a small reference program written independently of the training implementation. Such a program is often called a mathematical oracle: it provides a reference result for a correctness check. Here it compares a separate full projection at every computation with C3's shared token-side projections. A probe loss is an artificial scalar function of the outputs, used only to test their gradients. It is not the drafter's training loss. The check covers 108 cases in 64-bit

floating-point format (FP64). The largest absolute difference across the checked outputs, probe loss, and gradients is approximately 1.78×10^{-15}.

An earlier test used bfloat16 (BF16), a 16-bit floating-point format. It covered only normalization, Q/K/V projections, and RoPE. We measure relative-L_2 error by dividing the Euclidean norm of the difference by the norm of the reference. The output error was about 0.40–0.47%. The token-side root-mean-square normalization (RMSNorm) layer has a trainable scale parameter. Its gradient error was up to about 2.00%; this is not the gradient error of the token vector itself. These numbers describe that earlier implementation and its precision settings. They do not replace the full-model output, gradient, and optimizer-update comparisons in Section 5.3 and Appendix E.3.

The evaluated implementation combines token-side and hidden-side partials in FP32 to reduce intermediate rounding. The result can still differ bit by bit from one matrix multiplication on the concatenated inputs. Small local errors also do not prove that the methods give equivalent results across repeated long training runs. Activation checkpointing may rebuild a source table during backward recomputation, but its entries are still shared only within the same TTT unroll. End-to-end measurements include building and reading the tables, retaining FP32 partials, and replacing one wide matrix multiplication with separate token-side and hidden-side multiplications.

D Supervision Data

D.1 Data Sources and How We Measure Supervision

Figure 1 compares four training workloads: CNN, SQL, ShareGPT-20k, and DRAgent. We prepare CNN and SQL using HedgeSpec's data-processing pipeline and use public ShareGPT-20k conversations. For DRAgent, we use questions from DeepResearch Bench (Du et al., 2025) and the Researchy Questions set used by DeepResearchGym (Rosset et al., 2024; Coelho et al., 2025). We generate agent traces with the task-specific workflow based on Open Deep Research[2]. Each record contains the messages, tool results, and model outputs produced during the workflow. Listing 1 shows a real DRAgent sample generated from task 59 of DeepResearch Bench (Du et al., 2025). Its colors show which parts receive direct supervision.

A serialized sample is the token sequence formed by joining the messages and other text according to a task template. Its supervision ratio depends on how much of this sequence is selected as training targets. Long prompts, tool results, and intermediate interactions can add many context tokens without adding supervised positions. In DRAgent, system instructions, user prompts, and external tool results remain available as context but receive no direct loss. Only selected assistant-output tokens are supervised. Different tasks can therefore have different supervision ratios even when they use the same rule for selecting targets.

We process all sequences with the fixed Qwen3-8B tokenizer, Qwen chat template, and SpecForge rules for forming training sequences and marking target tokens. We exclude padding tokens. We also clear the last loss-mask position before counting, as required by the training target shift. For sample i, let L_i be its length after excluding padding. The binary mask $m_{i,t}$ is one if position t is directly supervised and zero otherwise. The direct-supervision ratio is

$$\rho_i = \frac{\sum_{t=1}^{L_i} m_{i,t}}{L_i}.$$

The Mean line gives each sample equal weight and averages its ratio ρ_i. The Micro line first adds the supervised-token counts across all samples and then divides by their total valid length. Longer samples therefore have more weight in Micro. These measurements describe the four workloads under this preprocessing setup; they are not estimates for all tasks.

[2]Open Deep Research: `https://github.com/langchain-ai/open_deep_research`.

[ORIGINAL QUESTION--UNSUPERVISED] In ecology, how do birds achieve precise location and direction navigation during migration? What cues and disturbances influence this process?

[SYSTEM--UNSUPERVISED] Break the research question into concise web search queries. Return at most three search queries.

[RESEARCH QUESTION--UNSUPERVISED] What specific environmental and celestial cues do migratory birds use to achieve precise navigation during long-distance migration?

[LLM TOOL QUERIES--SUPERVISED] migratory birds navigation environmental cues; celestial cues used by migratory birds; how do birds navigate during long-distance migration

[TOOL RESULTS--UNSUPERVISED]
[1] Orientation in migratory birds: time-associated relearning of celestial ...
Migratory birds use the geomagnetic field and celestial cues to identify their migratory direction.
[2] Magnetic information calibrates celestial cues during migration
Migratory birds use celestial and geomagnetic directional information to orient.
[3] Calibration of magnetic and celestial compass cues in migratory birds
Compass information includes the geomagnetic field, the sun, and skylight polarization.
[4] Migratory birds can extract positional information from magnetic ...
Birds use magnetic inclination and declination to determine direction during migration.

[INTERMEDIATE WORKFLOW--UNSUPERVISED] ... more tool results omitted ...

[LLM FINAL REPORT--SUPERVISED] Migratory birds employ a sophisticated array of sensory and cognitive mechanisms to achieve precise navigation during long-distance migration. Central to this process is detection of the Earth's magnetic field ...

Listing 1: **A real DRAgent sample generated from task 59 of DeepResearch Bench** (Du et al., 2025). The benchmark provides the original question. Gray text is input or workflow context; green text is tool output. Neither receives direct supervision. Orange text is selected model output used as a training target. Ellipses mark text omitted from this display only.

E Experimental Setup and Measurement Details

E.1 Workloads and Offline Training

We use separate experiments to measure training cost and trained-drafter quality. Table 7 gives the CNN and ShareGPT workloads used for the framework comparison and the C1–C3 component comparisons. Table 8 gives the DRAgent and ShareGPT splits used for the numerical and trained-quality comparisons in Table 2. Both setups use batch size 1. Their ShareGPT sample counts and training budgets differ, so results from the two setups should not be treated as measurements of the same workload. All training-cost comparisons use offline training: we run the target Transformer in advance and store its selected hidden states. Drafter training loads these stored features.

E.1.1 Training Recipe

Checkpoint and model. For the CNN and ShareGPT training-cost experiments in Table 7, SpecForge and our method use the same initialization checkpoint: `Tengyunw/qwen3_8b_eagle3`, revision `2a1059d51f622b8cad7d7d72840153ffea5488a0`. The target is `Qwen/Qwen3-8B`, revision `b968826d9c46dd6066d109eabc6255188de91218`. The drafter has one decoder layer, hidden size 4,096, MLP intermediate size 12,288, 32 query heads, eight K/V heads, and head dimension 128. Its input vocabulary has 151,936 entries and its output vocabulary has 32,000 entries. The configuration uses SiLU, RMSNorm with $\epsilon = 10^{-6}$, and RoPE with base 10^6; attention bias and attention dropout are disabled, and input embeddings and output-head weights are not tied.

Trainable parameters and optimization. The target model, including its normalization and output head, remains frozen. We train the drafter's target-feature projection, attention and MLP weights, normalization weights, and draft output head. In our implementation, the drafter input embedding weights are frozen and excluded from the optimizer. The optimizer is AdamW with FP32 master

Table 7: **Training-cost and component-ablation setup.**

Setting	CNN	ShareGPT
Samples	2,000	20,000
Optimizer steps	2,000	20,000
Independent complete runs	10	10
TTT length	7	7
Batch size	1	1
Target Model	Qwen/Qwen3-8B	Qwen/Qwen3-8B
Drafter configuration	`qwen3-8b-eagle3.json` (EAGLE-3)	`qwen3-8b-eagle3.json` (EAGLE-3)
GPU	One RTX 6000 Ada, 48 GB	One RTX 6000 Ada, 48 GB
Precision	BF16	BF16

Table 8: **Setup for numerical checks and trained-drafter evaluation.**

Setting	DRAgent	ShareGPT
Training split	1,800 samples	19,500 conversations
Test split	200 samples	500 conversations
Maximum sequence length	12,288 tokens	4,096 tokens
Epochs	5	1
Optimizer updates	9,000	19,500
Training seeds	10	10
TTT length	7	7
Batch size	1	1
Target Model	Qwen/Qwen3-8B	Qwen/Qwen3-8B
Drafter configuration	`qwen3-8b-eagle3.json` (EAGLE-3)	`qwen3-8b-eagle3.json` (EAGLE-3)
GPU	One RTX 6000 Ada, 48 GB	One RTX 6000 Ada, 48 GB
Precision	BF16	BF16

weights, zero weight decay, and a peak learning rate of 2×10^{-5}. Linear warmup covers the first 5% of the configured optimizer updates, followed by cosine decay to zero. Batch size and gradient accumulation are both one, and gradients are clipped to a global L_2 norm of 0.5 before the update.

TTT loss weighting and normalization. For one-based TTT step j, the loss weight is 0.8^{j-1}, with $j = 1, \ldots, 7$. The weighted step losses are summed without dividing by the sum of these weights. Within each step, masked token-level cross-entropies are summed and divided by the original, unpruned sequence-axis length of the loss tensor, since batch size is one. Masked-out positions contribute zero to the numerator but remain in this denominator; it is not the number of supervised or retained positions. When loss evaluation uses compact rows, it is rescaled to preserve this original denominator.

E.1.2 Framework Revisions and Peak-Memory Collection

Table 9 records the upstream base commits and allocator-peak collection points used in the CNN and ShareGPT training-cost experiments in Table 7. These commits identify the upstream base repositories. The runs also use local offline-data adapters and measurement wrappers.

Table 9: **Framework revisions and allocator-peak collection.**

Framework	Base commit and collection point
AngelSpec	`d3412bed231025ea85d35dbf5e0af44f1ac62a5b`; cumulative peaks read after optimizer steps inside the training actor.
SpecForge	`3d64e7a61f5fcc7f7d78ba6164c881f831943947`; cumulative peaks read after optimizer steps and after training returns.
TorchSpec	`6c042a87140a84d13839e341ece2c5c3ada918bc`; cumulative peaks read after optimizer steps inside the training actor.
Speculators	`35081a47fa5881504c35ad9a7be25af742b91e52`; cumulative peaks read after optimizer steps and after training returns.

The measurement wrappers for all four frameworks read `torch.cuda.max_memory_allocated()` and `torch.cuda.max_memory_reserved()` on the training device. The wrappers do not reset the peak counters, so these are cumulative allocator high-water marks rather than per-step allocation samples and can include initialization. Peaks are measured in the GPU-owning training process, not the launcher. Byte counts are divided by 2^{30} to obtain GiB. Device-level memory readings from NVML are not substituted for either allocator metric.

E.2 How We Average Repeated Runs

We run each framework or configuration in Table 1 and Panel (b) of Table 3 ten times on the complete workload. End-to-end (E2E) time is the arithmetic mean of the ten total training times. For Step wall, Forward, Backward, and C3 build, we first average the measured times over optimizer steps within each run, then average the ten run means. C3 build is the time used to construct the shared token-side projection tables. Timings can include one another and should not be added unless their measured intervals are disjoint.

Each run also gives one peak for tensor memory allocated by PyTorch and one peak for memory reserved by its caching allocator. Reserved memory includes unused blocks that PyTorch keeps for later use. We report the mean of the ten peaks for each measure. These two memory measures are not additive. Panel (a) of Table 3 instead counts positions and projection uses. These counts are fixed by the workload and configuration, so we compute them once over the complete workload. Section 5.1 defines the timing boundaries and units.

E.3 Numerical Checks and Trained-Drafter Evaluation

The numerical checks use the model and precision settings in Section 5.1. Both implementations load the same saved drafter weights to initialize training. This is the common warm-start checkpoint. They also use the same target features, first training batch, teacher probabilities, supervision

mask, and optimizer state. Comparing both implementations at the same parameter values isolates numerical differences caused by execution.

The metrics follow Section 5.3. Relative-L_2 error divides the norm of the difference by the norm of the corresponding SpecForge result. Output error compares the selected draft outputs used by the loss. Loss abs. gap is the absolute difference between the two scalar losses. All-gradient error joins the gradients of all trainable parameters into one vector before comparing them. Worst tensor rel.-L_2 instead reports the largest error for an individual parameter tensor, together with its name. Update error compares the changes made by the optimizer, $\Delta\theta = \theta_{\text{after}} - \theta_{\text{before}}$, rather than the full parameter values after the update.

Table 8 gives the training and test splits and the training budgets. The two implementations use ten paired training seeds: within each pair, they use the same random seed and training data. Panel (a) of Table 2 averages the numerical checks over these ten pairs. In Panel (b), Vanilla means the initial EAGLE-3 checkpoint before task-specific training. The SpecForge and our-method columns each average the test results from ten trained checkpoints. Every checkpoint uses the same held-out inputs and speculative-decoding settings.

Across the ten seed pairs, the mean all-gradient relative-L_2 error is 0.62% on DRAgent and 0.58% on ShareGPT. Gradient cosine similarity, which measures agreement in direction, is above 0.99998. The query-projection weight has the largest individual tensor error: 1.02% and 0.83%, respectively. Both implementations produce gradients for the same 13 trainable tensors in every pair. None of the checked gradients is missing or contains a non-finite value, such as an infinity or NaN.

Panel (b) of Table 2 shows comparable trained-drafter behavior for our method and SpecForge. The percentages below use the displayed means and the named baseline as the denominator. Relative to SpecForge, our method's tokens advanced per verification differ by at most 0.78%, and its accepted draft tokens per verification differ by at most 1.06%. Generation throughput, measured in tokens per second, differs by at most 0.67%. Both trained implementations outperform Vanilla. Compared with Vanilla, our method increases accepted drafts per verification by 54.05% on DRAgent and 15.79% on ShareGPT. Throughput increases by 17.04% and 9.76%, respectively. These results show that the execution changes preserve the training benefits measured in this setup.

E.4 Training-Loss Curves Across the Full Runs

Figure 5 compares training loss over the full runs. At each optimizer update, we first average the loss over the ten training seeds. We then smooth both curves with the same 100-update moving mean: each plotted point averages the current and up to 99 preceding update means. No recorded update is removed.

During the first 50 updates, the largest absolute loss gaps are 1.06×10^{-3} on DRAgent and 5.59×10^{-3} on ShareGPT. Over all 9,000 and 19,500 updates, the mean absolute gaps are 0.00719 and 0.00128. For the final 10% of updates, we also weight each update's loss by its number of supervised token positions and divide by the total of those counts. These token-weighted mean losses differ by 0.000130 and 0.000301. The results describe training loss from the paired runs. They do not measure held-out loss or formally test whether the methods give equivalent results across repeated training runs.

F Definitions for the Component Comparisons

The component comparisons in Table 3 measure the effect of enabling C1, C2, and C3. They use the complete CNN and ShareGPT training-cost workloads in Table 7, with SpecForge as the baseline. Panel (b) follows Appendix E.2: each configuration has ten complete runs per workload. Panel (a) counts which positions and projections are used. These counts depend on the fixed data and execution rules, so they are computed once over each complete workload.

Supervision is the total number of directly supervised positions divided by $\sum_s L_s$, where L_s is the valid length of sample s. Public K/V coverage uses the same denominator, but counts positions whose public K/V are built. Keeping full public context therefore gives complete public K/V coverage, even when fewer queries are executed.

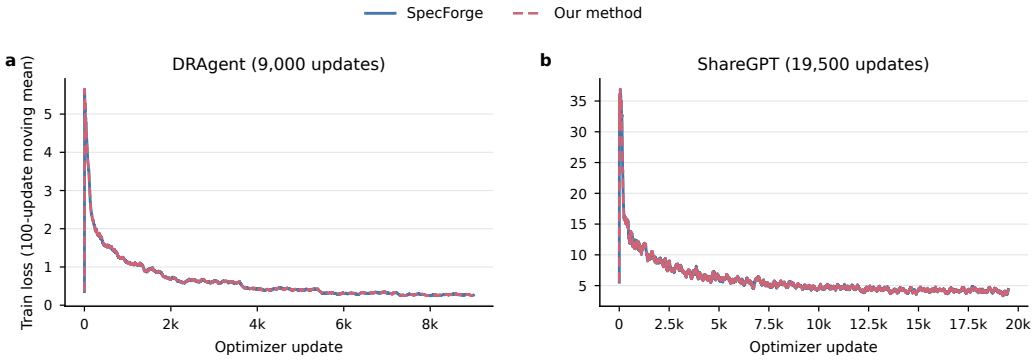

Figure 5: **Training-loss curves for ten paired seeds.** Each pair starts from the same state. At every update, we average the loss over ten seeds and then apply a 100-update moving mean. The curves include all 9,000 DRAgent updates and 19,500 ShareGPT updates. Updates within a run are not independent repetitions of the experiment.

The source-position reuse factor counts how often token-side projection work would be repeated without C3. For $X \in \{Q, KV\}$, it is

$$\text{Reuse}_X = \frac{\text{token-side partial evaluations without C3}}{\text{distinct source positions materialized by C3}}.$$

The numerator counts all uses that would require a separate token-side projection without C3. The denominator counts the source positions for which C3 stores a reusable partial. We count distinct positions within each sample, then sum the counts over samples; there is no sharing across samples. One K/V evaluation means computing both the K and V partials for one source position. The reuse factor is a ratio of computation counts, not a measured speedup.

Panel (b) uses the timing and memory definitions in Section 5.1. Each percentage reduction names its reference, either SpecForge or C1. The reduction is the difference from that reference divided by the reference value.

G Extended Related Work

This section expands the related-work discussion in Section 2.

G.1 Training Drafters for the Target Model and Tasks

Several studies train drafters to better match the target model or the tasks seen during use. EAGLE-3 uses target-model features and TTT to train recurrent token predictions (Li et al., 2025). HASS studies differences between training and decoding contexts and objectives (Zhang et al., 2024). DistillSpec uses the target model to teach the drafter, a process called knowledge distillation (Zhou et al., 2023). Domain-draft training studies task-domain data and distillation (Hong et al., 2025). Online Speculative Decoding adapts drafters to queries observed during use (Liu et al., 2023). Aurora combines this adaptation with serving the model (Wang et al., 2026a). These studies motivate task-specific drafter training. We study the execution cost of this training while keeping its selected targets and loss function.

G.2 Training Infrastructure and Efficiency

Other work reduces the time and memory needed to prepare drafters. SpecForge separates target-model and drafter execution, combines forms of parallel training, skips unused attention blocks with FlexAttention, and reduces gradient memory (Li et al., 2026). Our implementation builds on this training pipeline. TorchSpec transfers features between separate generation and training workers through the Mooncake transfer system (TorchSpec team & Mooncake team, 2026). Speculators and Model Optimizer provide tools for drafter training and deployment (Speculators contributors, 2026; NVIDIA, 2026). FastDraft also studies efficient drafter preparation (Zafrir et al., 2024).

PARD and P-EAGLE reduce parallel-drafter training costs by dropping tokens during adaptation or partitioning sequences, respectively (An et al., 2025; Hui et al., 2026). P-EAGLE keeps the dependencies between prediction depths and the required context from depth zero when selecting sequence parts. C1 instead keeps the recurrent EAGLE-3 computations needed by a fixed set of supervised targets. LK Losses changes the training loss to better support draft acceptance (Samarin et al., 2026). Our method keeps the existing loss against the target model's token probabilities.

G.3 Structured Training Attention

Several methods split attention into shared context and history specific to one branch or path. EAGLE-3 describes public K/V shared across paths and private K/V from earlier steps of the same path (Li et al., 2025). Its private connections join corresponding positions across steps and can be computed with individual dot products. AngelSpec computes public attention with FlexAttention, computes private attention with dot products, and combines the results using LSE values (Liu et al., 2026a; Tencent, 2026). We call this its partition-and-LSE implementation. It also provides the backward computation needed for training. Hydragen separates shared-prefix attention during inference, and FlashInfer supports combining separately computed attention states (Juravsky et al., 2024; Ye et al., 2025). Online Draft Co-Training combines causal main-sequence attention with local-branch attention during training (Wang et al., 2026b).

Our method builds on these attention splits. In the AngelSpec EAGLE-3 implementation we inspected, queries cover full-sequence positions and the public stage stores outputs and LSE values for a later private merge. C1 selects queries according to whether their recurrent computations can reach a supervised loss. Fused Attention for Retained Queries (C2) directly accepts this smaller, possibly non-contiguous query set while keeping full public K/V. Both attention partitions update one online-softmax state within a single forward kernel, avoiding the separate public-only outputs and LSE values. The final output and LSE are saved for backward. Appendix B.1 compares C2 with an adapted reference that uses the same retained queries, isolating the effect of the attention implementation.

G.4 Projection Reuse and Training Memory

Recurrent neural network (RNN) optimizations separate the input projection from the recurrent-state projection and process available inputs together (Appleyard et al., 2016). In EAGLE-3 training, different TTT paths and TTT steps may read the same source position even though their hidden states differ. C3 computes that position's token-side projection once and shares it across these computations. It uses separate Q and K/V source maps and adds gradients from every use of a shared partial. FlashAttention reduces attention memory traffic, while activation checkpointing saves memory by recomputing intermediate results (Dao et al., 2022; Chen et al., 2016). We use these techniques for the retained computations and include preparation and recomputation in our measurements.

H SCOPE OF THE DIRECT IMPLEMENTATION COMPARISON

Our code comparison asks two separate questions: which keys can each query read, and which recurrent computations does the system execute? In the AngelSpec EAGLE-3 partition-and-LSE implementation we inspected, the recurrent backbone runs before selecting loss positions, and queries remain arranged over the full sequence. C1 selects the required starting positions before running the backbone, while preserving all TTT-step-1 public K/V.

AngelSpec uses shared normalization and a custom backward pass, including the summation of gradients from query heads that share a K/V head. C2 accepts C1's smaller query set with full public K/V and processes both attention partitions in one forward kernel. It saves the final output and LSE for backward while avoiding public-only outputs and LSE values for a separate merge. Its backward uses multiple kernels. The comparison in Appendix B.1 holds the retained queries and the other execution settings fixed to measure the effect of this attention design.

P-EAGLE partitions sampled positions across prediction depths in parallel-drafter training while keeping required dependencies (Hui et al., 2026). C1 derives the recurrent computations needed by a fixed set of supervised targets and retains full TTT-step-1 public K/V. Online Draft Co-Training also combines main-sequence and branch attention for training (Wang et al., 2026b). Our code comparison uses the NeMo-RL pull request at commit `72ed478`, which had not been merged at the time of inspection.

I Full Supervision-Density and Layout Sensitivity Results

Tables 10–17 extend the synthetic-data step-wall comparison in Table 4 to 8,192 and 12,288 tokens with TTT lengths 3, 5, and 7, and to 18,432 tokens with TTT lengths 3 and 5. Each table covers one length and one TTT length. Densities of 10%, 25%, 50%, and 75% use Tail, Uniform, and Random layouts; unavailable cells are left blank. At 100% density, all layouts coincide and are shown as one Full row.

All configurations use synthetic training inputs and 2,000 optimizer steps. We compare SpecForge with Ours using mean complete optimizer-step wall time over these steps. Step-wall speedup is SpecForge's mean step wall divided by our method's mean step wall. Each allocator-peak reduction is $100(M_{\mathrm{SpecForge}} - M_{\mathrm{Ours}})/M_{\mathrm{SpecForge}}$, using allocated or reserved peak separately from the same runs. Ratios and percentages are displayed to two decimal places.

Table 10: **Supervision-density and layout sensitivity at 8,192 tokens and TTT length 3.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	2.68	36.60	27.08
10	Uniform	2.56	36.60	22.41
10	Random	2.61	36.60	25.31
25	Tail	2.08	36.60	21.65
25	Uniform	1.81	36.60	19.48
25	Random	1.95	36.60	20.05
50	Tail	1.52	34.57	22.90
50	Uniform	1.34	28.67	20.05
50	Random	1.46	31.60	20.89
75	Tail	1.32	22.72	15.92
75	Uniform	1.18	19.78	14.58
75	Random	1.27	20.86	14.75
100	Full	1.12	10.88	8.62

Table 11: **Supervision-density and layout sensitivity at 8,192 tokens and TTT length 5.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	3.50	47.07	44.45
10	Uniform	2.92	47.07	43.86
10	Random	3.21	47.07	44.05
25	Tail	2.47	47.07	43.63
25	Uniform	2.01	40.07	40.98
25	Random	2.18	43.85	41.03
50	Tail	1.88	37.52	44.19
50	Uniform	1.41	27.46	40.57
50	Random	1.62	30.56	41.52
75	Tail	1.29	23.68	28.33
75	Uniform	1.17	18.68	21.41
75	Random	1.23	22.46	23.27
100	Full	1.12	9.89	16.35

Table 12: **Supervision-density and layout sensitivity at 8,192 tokens and TTT length 7.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	4.02	54.79	52.49
10	Uniform	3.10	52.68	46.82
10	Random	3.36	53.51	48.32
25	Tail	2.62	54.79	47.46
25	Uniform	1.96	38.61	43.21
25	Random	2.15	41.25	44.68
50	Tail	1.94	39.67	49.18
50	Uniform	1.49	26.47	39.10
50	Random	1.67	30.51	42.86
75	Tail	1.44	24.37	29.45
75	Uniform	1.26	17.80	26.02
75	Random	1.32	20.74	26.02
100	Full	1.11	9.14	13.59

Table 13: **Supervision-density and layout sensitivity at 12,288 tokens and TTT length 3.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	3.08	51.50	41.91
10	Uniform	2.84	51.50	40.37
10	Random	2.93	51.50	40.98
25	Tail	2.19	51.50	40.91
25	Uniform	1.77	48.19	39.30
25	Random	1.98	48.92	39.65
50	Tail	1.65	39.71	36.22
50	Uniform	1.36	32.87	29.33
50	Random	1.49	35.61	31.92
75	Tail	1.38	26.06	23.20
75	Uniform	1.19	22.68	19.52
75	Random	1.27	23.54	20.97
100	Full	1.12	12.45	16.33

Table 14: **Supervision-density and layout sensitivity at 12,288 tokens and TTT length 5.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	3.70	60.48	47.08
10	Uniform	2.86	60.13	44.53
10	Random	3.25	60.13	44.97
25	Tail	2.68	57.56	45.76
25	Uniform	1.93	44.91	34.93
25	Random	2.27	46.87	40.25
50	Tail	2.01	42.06	38.90
50	Uniform	1.51	30.74	27.82
50	Random	1.69	36.21	30.96
75	Tail	1.48	26.53	23.45
75	Uniform	1.20	20.89	17.32
75	Random	1.28	22.57	18.21
100	Full	1.15	11.06	5.29

Table 15: **Supervision-density and layout sensitivity at 12,288 tokens and TTT length 7.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	4.39	66.82	51.71
10	Uniform	3.10	59.55	45.87
10	Random	3.56	63.09	46.06
25	Tail	2.80	54.80	47.50
25	Uniform	2.08	38.60	43.20
25	Random	2.34	43.33	43.90
50	Tail	2.17	39.70	49.20
50	Uniform	1.58	26.50	29.10
50	Random	1.76	28.72	31.94
75	Tail	1.55	24.40	28.40
75	Uniform	1.24	16.80	18.14
75	Random	1.35	18.18	19.88
100	Full	1.18	11.92	13.26

Table 16: **Supervision-density and layout sensitivity at 18,432 tokens and TTT length 3.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	3.12	64.14	54.59
10	Uniform	2.98	64.14	54.31
10	Random	3.04	64.14	57.79
25	Tail	2.10	59.16	53.09
25	Uniform	1.88	53.47	50.54
25	Random	1.92	55.65	51.04
50	Tail	1.73	44.03	41.12
50	Uniform	1.48	36.44	33.74
50	Random	1.56	40.49	35.03
75	Tail	1.43	28.90	21.82
75	Uniform	1.24	25.14	21.80
75	Random	1.32	29.09	23.31
100	Full	1.12	13.81	8.12

Table 17: **Supervision-density and layout sensitivity at 18,432 tokens and TTT length 5.**

Density (%)	Layout	Step-wall speedup (×) SpecForge/Ours	Allocated peak reduction (%) vs. SpecForge	Reserved peak reduction (%) vs. SpecForge
10	Tail	3.98	71.35	58.36
10	Uniform	3.01	66.60	54.86
10	Random	3.43	68.18	55.95
25	Tail	2.75	62.60	55.43
25	Uniform	2.01	48.82	43.92
25	Random	2.19	52.70	52.71
50	Tail	1.91	45.72	37.84
50	Uniform	1.46	33.43	28.61
50	Random	1.62	42.86	32.36
75	Tail	1.53	28.85	22.73
75	Uniform	1.21	22.72	15.59
75	Random	1.36	24.03	20.58
100	Full	1.13	12.01	7.51