APEX: SPECULATE SMARTER, NOT DEEPER

Anonymous authors
Paper under double-blind review

ABSTRACT

Speculative decoding is a leading approach for reducing large language model inference latency, but existing systems still treat speculation as a mostly static configuration: choose a proposal mechanism, choose a depth, and apply it uniformly. This static treatment is mismatched to dynamic generation, where the value of speculation changes across prompts, prefixes, entropy regimes, repetition structure, verifier outcomes, and proposal cost. When the first rejection occurs early, deeper speculation does not accelerate decoding; it simply converts compute into wasted draft tokens. We introduce **APEX**, a learned controller that balances acceleration and draft-token waste through request-level expert selection and block-level depth adaptation. **APEX-Router** selects the speculative expert for each request, choosing among existing baselines like EAGLE-3, n-gram, and draft-model speculation, while **APEX-Depth** adapts the number of draft tokens mid-generation at each verifier block. Its key technical idea is to model accepted draft length as censored survival feedback. Instead of predicting a coarse acceptance rate, **APEX** learns position-wise rejection hazards and block-level cost, then selects the action that maximizes expected accepted progress per unit cost. We instantiate **APEX** inside vLLM and evaluate it with Qwen3-8B across several workloads. **APEX** improves the speed-waste tradeoff over strong fixed speculative configurations, achieving up to 5.24× speedup over autoregressive decoding and reducing wasted speculative tokens to 40% compared against fixed-depth baselines. **APEX** shows that the next step in speculative decoding is not deeper drafting, but adaptive control.

1 INTRODUCTION

Large language models (LLMs) increasingly serve interactive and long-form workloads such as code generation, mathematical reasoning, and conversational assistance. In these settings, inference latency is not merely a backend cost, it directly affects user-perceived responsiveness and the feasibility of deploying capable models at scale (Kwon et al., 2023b; Agrawal et al., 2024). Yet acceleration must preserve the behavior of the target model. The systems challenge is therefore not simply to generate more tokens per second, but to convert additional computation into verified output progress without changing the target model's generation semantics.

Autoregressive decoding (Schuurmans et al., 2024) A.1 is inherently sequential: each token requires a target-model step conditioned on the full prefix. Speculative decoding (SD) (Xia et al., 2023) A.2 drafts multiple tokens using a cheaper proposal mechanism, then verifies them with the target model. Accepting a long draft prefix advances generation by multiple positions per verifier step, but high speedup can still coexist with substantial draft-token waste. Evaluating SD therefore requires measuring both latency reduction and draft-token utilization.

Whether speculation is beneficial depends on the decoding regime (Yan et al., 2025). Neural speculators such as EAGLE-3 (Li et al., 2025a) can be effective when the target distribution is predictable; retrieval-based methods such as n-gram speculation (He et al., 2024) can be efficient when the context contains reusable structure; and draft-model speculation can produce plausible continuations whose acceptance gains may nevertheless be offset by proposal latency. Similarly for speculation depth, a large depth offers greater potential progress when drafts survive, but amplifies wasted work when the first rejection arrives early.

The value of speculation depends jointly on the proposal mechanism, draft depth, and decoding state. A mechanism that proposes tokens cheaply may still discard a large fraction of its drafts,

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.

while a more accurate mechanism may incur enough proposal latency to offset its acceptance gains. Selecting an appropriate mechanism and depth requires accounting for the progress and cost of verification. This motivates our central research question:

Can speculative decoding adapt its proposal mechanism and depth to maximize verified progress per unit cost as decoding evolves?

We address this question with APEX, a learned controller that selects a proposal expert and initial depth for each request, then adapts depth to the evolving decoding state. APEX-Router makes the request-level decision, while APEX-Depth uses causal decoding signals and prior verifier feedback at each block. By learning position-wise rejection hazards and block costs, APEX scores candidate depths using a utility that balances throughput, accepted progress, and wasted draft tokens.

Contributions. Our work makes the following contributions:

1. **Empirical motivation for adaptive speculative control.** We characterize how proposal mechanisms and depths trade speed against draft-token waste across workloads, entropy, repetition, and proposal costs. These observations motivate selecting experts and depths using decoding-state information.
2. **A survival-cost formulation of speculative control.** We model accepted draft length using rejection hazards and censored feedback, yielding a cost-aware objective that balances accepted progress, latency, and wasted tokens.
3. **APEX, a hierarchical controller for heterogeneous speculation.** We introduce a two-level controller in which APEX-Router selects an expert per request, while APEX-Depth causally adapts the speculation depth at each verifier block using inference-time signals.
4. **Implementation and evaluation.** We instantiate APEX in vLLM with Qwen3-8B for training and evaluation. APEX achieves up to $5.24\times$ speedup over autoregressive decoding and takes down token wastage to upto 40% relative to fixed-depth baselines.

Broader outlook. A proposal mechanism is valuable only when its accepted progress outweighs the cost of producing and verifying its drafts, and that balance changes as generation evolves. In APEX, request-level routing selects a proposal mechanism, while block-level depth control adjusts how much work that mechanism performs as decoding evolves. Separating these decisions allows existing proposal algorithms to operate under a shared control framework. The resulting speed–waste trade-offs depend on the workload and controller profile, making both latency and discarded draft tokens necessary evaluation outcomes. The next step in SD is therefore not uniformly deeper drafting, but adaptive, cost-aware speculation.

2 Background and Related Work

2.1 Speculative Mechanisms

Early SD methods use an auxiliary draft model to propose tokens for target-model verification (Xia et al., 2023; Leviathan et al., 2023). Followup works improve drafting through knowledge distillation, specialized architectures, or objectives aligned with target-model acceptance (Bhansali & Heck, 2025; Zou et al., 2026). Improving an individual proposal mechanism does not by itself determine when that mechanism is preferable to alternatives with different proposal costs and acceptance behavior. A fixed choice can therefore retain a costly or poorly matched speculative path even when another mechanism offers a more favorable speed–waste trade-off.

A second line of work reduces reliance on separate draft models. Medusa adds decoding heads to the target model for parallel verification of tree-structured continuations (Cai et al., 2024), while SpecInfer constructs token trees for parallel candidate verification (Miao et al., 2024). EAGLE-style methods instead use target-model hidden states to build lightweight neural speculators (Li et al., 2025b;a). Their gains, however, still depend on proposal survival and cost.

A third class uses retrieval or surface repetition instead of neural drafting. N-gram and suffix-based methods copy continuations from the prompt, generated prefix, or external text, making them cheap

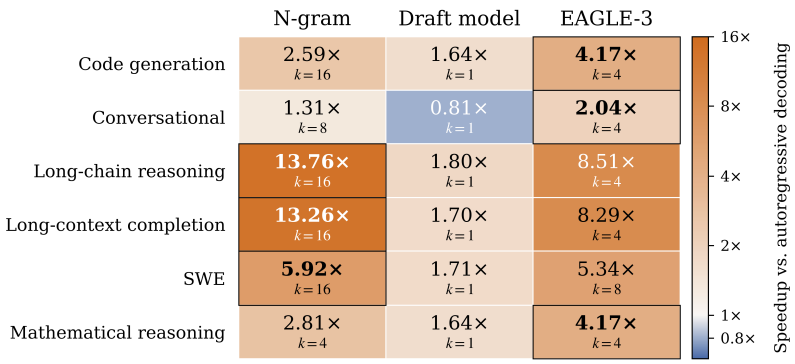


Figure 1: **Speedup over autoregressive decoding** across six workloads and three speculative mechanisms. Each cell reports highest speedup and its depth k; boxes mark the best method per workload.

and effective in repetitive regimes but brittle on novel continuations (Hu et al., 2024). This creates a complementary tradeoff: neural speculators can better model semantic continuation, while retrieval-based speculators can be faster when repeated structure is available.

APEX is orthogonal to these mechanisms; it treats mechanisms like EAGLE-3, n-gram, and draft-model speculation as candidate experts and learns which of them to use and how deeply to speculate in each decoding state. APEX can thus choose among mechanisms with different proposal costs and acceptance behavior while retaining their existing proposal and verification algorithms.

2.2 Adaptive Speculative Decoding

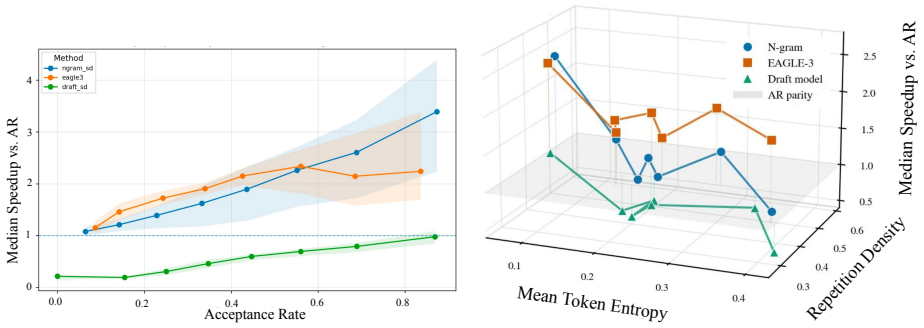
Dynamic Speculation Lookahead shows that using the same speculation lookahead for all decoding iterations is suboptimal and proposes dynamically selecting the number of draft tokens during generation (Mamou et al., 2024). AdaEAGLE (Zhang et al., 2024)explicitly models adaptive draft structures and uses a lightweight draft length predictor to choose the number of draft tokens for EAGLE-style speculation. BanditSpec (Hou et al., 2025) formulates speculative hyperparameter selection as a multi-armed bandit problem and adapts the configuration online without training a controller. DSDE (Yang et al., 2025) studies dynamic speculation length in serving settings using KLD-based stability signals and adaptive caps. Production systems such as vLLM also support dynamic SD through manually specified schedules that map concurrency or batch-size ranges to a chosen number of speculative tokens (Kwon et al., 2023a).

These studies show that a single fixed depth (k) can limit inference acceleration as acceptance behavior and serving conditions change. They motivate a broader control problem: selecting a proposal mechanism together with depths that balance verified progress, runtime, and draft-token waste. APEX addresses this problem through request-level expert selection and block-level depth adaptation, using learned rejection hazards and runtime signals to score candidate depths.

3 Empirical Motivation: Fixed Speculation Is Not Enough

We ask whether SD is well served by a fixed proposal mechanism and speculation depth. Using Qwen3-8B in vLLM, we compare autoregressive decoding with EAGLE-3, n-gram, and draft-model speculation across depths $k \in \{1, 2, 4, 8, 16\}$ and diverse workloads spanning code, mathematics, long-context and long-chain reasoning, conversation, and SWE-style tasks. Workload details and additional results appear in Section 5 and Appendix A.8.

Speculative performance is regime-dependent. Figure 1 shows that the best SD configuration varies across workloads. EAGLE-3 is broadly strong, while n-gram speculation benefits from cheap, reliable context reuse in repetitive long-context settings. Draft-model speculation often fails to translate acceptance into speedup due to auxiliary-model overhead (Figure 2a). These differences motivate adaptive speculator selection as regimes change.



(a) Acceptance and speedup

(b) Entropy–repetition vs speedup

Figure 2: **Online signals associated with SD performance** (a) Median speedup over autoregressive decoding versus acceptance rate, with shaded variability. (b) Median speedup versus token entropy and repetition density; the gray plane marks autoregressive performance.

Speculation depth changes the speed–waste trade-off. The aggregate sweep in Table 3 illustrates why greater depth need not translate into proportionate acceleration. Increasing n-gram depth from $k = 8$ to $k = 16$ raises speedup from $3.59\times$ to $4.22\times$, while mean waste increases from 63.56% to 79.90%. For EAGLE-3, increasing depth from $k = 4$ to $k = 16$ reduces mean speedup from $2.35\times$ to $1.37\times$ and raises mean waste from 66.25% to 97.67%. These different responses motivate choosing depth according to the active mechanism and expected verification outcome.

Observable signals predict when speculation works. The variation is predictable: EAGLE-3 speedup decreases as token entropy increases, while n-gram speedup increases with prefix repetition density, shown in Figure 2b. This aligns with the mechanisms: neural speculative prediction becomes riskier when the target distribution is uncertain, whereas n-gram speculation becomes more effective when recent context contains reusable continuations. These signals are available during decoding, suggesting that a controller can adapt using causal online features without workload labels.

Accepted length is a survival quantity. For a depth-k block, first rejection at position r leaves $r-1$ accepted draft tokens and discards the rest; full acceptance reveals only survival through depth k. Unlike aggregate acceptance statistics, position-wise rejection hazards capture how survival and accepted progress vary with depth.

4 APEX: Breathing Adaptivity into Static Speculation

We propose **APEX**, a learned online controller that selects a proposal mechanism at request granularity and adjusts speculation depth at block granularity. Its learned utility balances throughput and accepted progress against discarded draft tokens. **APEX** operates at two coupled timescales. **APEX-Router** selects a speculative expert—EAGLE-3, n-gram, or draft-model speculation—at the request level, while **APEX-Depth** adapts depth k at the block level. This separation reflects systems constraints: expert selection affects model placement, batching, and worker routing, whereas depth can change more frequently within the selected path.

4.1 Problem Formulation

Let p_θ denote the target language model and let $m \in \mathcal{M}$ denote a proposal mechanism, such as n-gram SD, EAGLE-3, or draft-model speculation. Given a generated prefix $x_{\le t}$, mechanism m proposes a draft block of length k $\hat{x}_{t+1:t+k} \sim q_m(\cdot \mid x_{\le t})$, where q_m is its induced proposal distribution. The target model then verifies the draft block. Let $L_t \in \{0, \dots, k\}$ be the number of accepted draft tokens in verifier block t. If $L_t = k$, the full block is accepted; if $L_t < k$, the verifier rejects at position $L_t + 1$. We define the discarded draft-token count and waste fraction as:

$$R_t = k - L_t, \qquad W_t = \frac{R_t}{k} \tag{1}$$

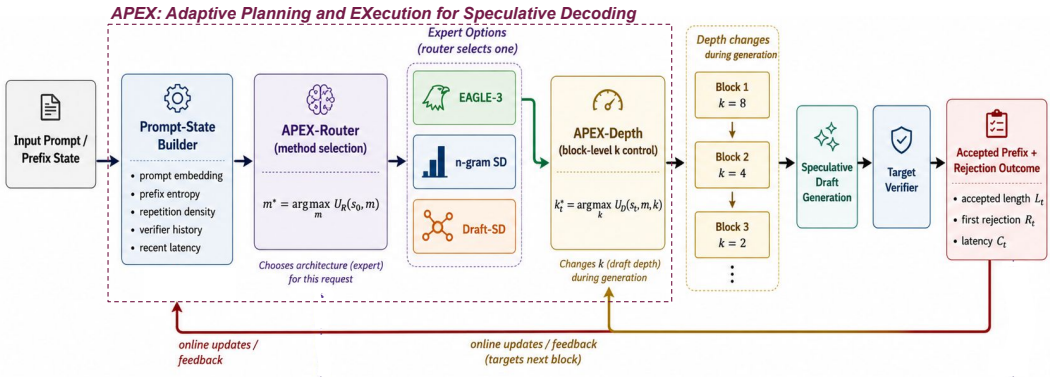


Figure 3: APEX **overview.** The router selects a speculative expert and initial depth from prompt features. At each block, the depth controller uses decoding state and prior verifier feedback to select the next depth; observed acceptance and runtime then update the controller online.

Here, R_t includes the first rejected token and the discarded suffix, and W_t is the fraction of drafted tokens that are not accepted. Since W_t measures token utilization rather than wasted computation, we measure latency separately to assess the speed–waste trade-off.

Let C_t denote block-t wall-clock latency, including drafting and verification, and TPS_t the executed action's throughput. APEX uses both to learn a block-level utility balancing accepted progress, runtime, and speculative waste. In our experiments, $K_{\max} = 16$, and the candidate depths form a finite set $\mathcal{K} \subseteq 1, \ldots, K_{\max}$. APEX selects the next speculation depth by maximizing the learned block-level utility:

$$k_t^\star = \arg\max_{k \in \mathcal{K}} \widehat{U}_D(s_t, m, k), \tag{2}$$

where s_t is the causal state available before verifier block t, m is the active speculative mechanism, and $\widehat{U}_D$ is the learned utility predicted by APEX.

4.2 APEX-Router

APEX-Router selects the speculative expert expected to provide the best speed–cost tradeoff for each prompt before generation, over the candidate set $\mathcal{M} = \{\text{EAGLE-3}, \text{n-gram SD}, \text{Draft-SD}\}$.

For each request, the pre-generation state (s_0) includes the prompt embedding, length, token and repetition statistics, entropy-related features, and decoding configuration (information available before decoding begins), as detailed in Appendix A.3. The router scores initial actions $a = (m, k_0)$, where $m \in \mathcal{M}$ is the speculative mechanism and k_0 the initial depth. It predicts:

$$\widehat{U}_R(s_0, a) = f_{\theta_R}(\phi_R(s_0, a)), \tag{3}$$

where ϕ_R is the router feature map and f_{θ_R} is a neural scoring function. The selected action is

$$a^\star = (m^\star, k_0^\star) = \arg\max_{a \in \mathcal{A}_R} \widehat{U}_R(s_0, a). \tag{4}$$

The router is trained from completed request-level traces rather than workload labels. For each request r and candidate action a, the dataset records throughput, latency, accepted-token progress, and speculative waste, which are mapped to the empirical request-level utility

4.3 APEX-Depth

APEX-Depth adapts speculation depth to balance accepted progress, latency, and draft-token waste. Its operation begins with a speculative method and initial depth selected through request-level routing. For request r and candidate action $a = (m, k_0)$, the offline routing score is

$$U_R(r, a) = \Psi_R\big(\mathrm{TPS}_{r,a}, \mathrm{Latency}_{r,a}, \mathrm{Accepted}_{r,a}, \mathrm{Waste}_{r,a}\big)\,. \tag{5}$$

Algorithm 1 APEX Online Speculative Decoding

Require: Prompt x, target model p_θ, mechanisms $\mathcal{M}$, depths $\mathcal{K}$
1: $(m, k_1) \leftarrow \pi_R(\phi_R(x))$, $\mathbf{h}_0 \leftarrow \emptyset$, $y \leftarrow \emptyset$
2: **for** $t = 1, 2, \ldots$ until generation completes **do**
3: $\quad s_t \leftarrow \mathrm{State}(x, y, m, k_t, \mathbf{h}_{t-1})$
4: $\quad d_t \leftarrow \mathrm{Draft}(m, k_t; s_t)$
5: $\quad (\Delta y_t, o_t) \leftarrow \mathrm{Verify}(p_\theta, d_t)$, where $o_t = (L_t, R_t, W_t, C_t)$
6: $\quad y \leftarrow y \,\|\, \Delta y_t, \quad \mathbf{h}_t \leftarrow \mathrm{Update}(\mathbf{h}_{t-1}, o_t)$
7: $\quad s_{t+1} \leftarrow \mathrm{State}(x, y, m, \mathbf{h}_t)$
8: $\quad k_{t+1} \leftarrow \arg\max_{k \in \mathcal{K}} \widehat{U}_D(\phi_D(s_{t+1}, m, k))$
9: **end for**

The scoring rule Ψ_R defines training targets; at inference, the router uses only the initial state s_0 to predict $\widehat{U}_R(s_0, a)$. The selected action $a^* = (m^*, k_0^*)$ determines the speculative decoding path and initializes its depth. APEX-Depth then controls subsequent depths through the following steps.

Observe the decoding context. Before block t, the controller receives a causal state s_t containing request features, the current prefix, generation progress, and feedback from earlier verifier blocks (Appendix A.3). It does not observe the acceptance outcome of the block being predicted. For each candidate depth $k \in \mathcal{K}$, it constructs a shared representation:

$$z_t(k) = \phi_D(s_t, m^*, k), \qquad h_t(k) = g_\theta(z_t(k)). \tag{6}$$

Here, ϕ_D standardizes numerical features, embeds categorical variables, and appends the prompt representation. The shared encoder therefore evaluates each depth in the context of the current generation state.

Learn the value of each depth. A utility head uses this representation to predict $\widehat{U}_D(s_t, m^*, k)$, the score used for depth selection. Three auxiliary heads help it capture the execution outcomes underlying this decision. The *survival head* predicts conditional rejection hazards $\hat{q}_{t,j}(k)$ across draft positions, determining the survival curve and expected accepted length. The *cost head* predicts log block latency $\widehat{\log C_t(k)}$, while the *throughput head* predicts $\widehat{\log(1 + \mathrm{TPS}_t(k))}$. Appendix A.4 provides their detailed formulation.

These auxiliary tasks ground the shared representation in measurable outcomes, while the utility head learns the deployed selection score. This distinction matters because accepted length alone does not determine efficiency: deeper speculation may increase accepted progress but incur greater runtime and waste, whereas shallow speculation may limit waste without sufficiently advancing generation.

Select, execute, and update. The controller chooses the highest-scoring depth, and executes the corresponding speculative block using equation 7. Its observed outcomes update the state for the next decision, allowing depth to respond to changes in acceptance behavior and execution cost throughout generation.

$$k_t^* = \arg\max_{k \in \mathcal{K}} \widehat{U}_D(s_t, m^*, k), \tag{7}$$

4.4 Trace-Derived Training Targets

We can train APEX from offline collected speculative decoding traces. Each training example corresponds to an observed verifier block under a particular state-action pair $(s_t, m, k) \longrightarrow (L_t, R_t, W_t, C_t, \mathrm{TPS}_t)$. The input contains only information available before executing the block, while the target contains the outcome observed after verification. This makes the controller causal and deployable. For each block, APEX constructs a scalar utility target:

$$U_t = \log(\mathrm{TPS}_t) + \alpha_A \log(L_t + \epsilon) - \alpha_W W_t - \alpha_G G_t - \alpha_M M_t \tag{8}$$

Here, ϵ ensures numerical stability, G_t is regret relative to the best observed method-depth action for the same request in the offline trace dataset, and M_t is regret relative to the best depth within the same method. We refer details of the regret and utility targets to A.5 and A.6.

Table 1: **Overall decoding performance**. APEX is compared against autoregressive and fixed speculative baselines (averaged over test set). Best value is in **bold** & second-best is underlined AR=1.00×

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 8$	1.94 ×	87.11	2.23
EAGLE-3	$k = 16$	1.37 ×	97.67	1.41
n-gram SD	$k = 8$	3.59 ×	63.56	5.64
n-gram SD	$k = 16$	4.22 ×	79.90	5.28
Draft-SD	$k = 8$	0.73 ×	52.46	1.39
Draft-SD	$k = 16$	0.50 ×	68.98	0.72
APEX-S	adaptive	**4.27** ×	55.30	**7.73**
APEX-E	adaptive	1.83 ×	63.69	2.87
APEX-B	adaptive	3.27 ×	**47.17**	6.93

Together, APEX combines request-level method selection with block-level depth adaptation, learning from execution traces to balance accepted progress, runtime, and draft-token waste. At inference, these learned decisions use only causal state information and adapt as generation evolves. We next evaluate whether this approach delivers substantial acceleration with improved draft-token utilization across workloads, relative to baseline SD configurations.

5 Results and Discussion

5.1 APEX Variants

We train three APEX variants with identical architectures, features, candidate-depth interfaces, data, and deployment procedures; only the utility weights in equation 8 differ, producing distinct speed–waste operating points.

APEX-S (Speed) uses $(\alpha_A, \alpha_W, \alpha_G, \alpha_M) = (0.20, 0.20, 0.20, 0.15)$ and places greater relative emphasis on throughput competitiveness. The default APEX-B (Balanced) uses $(0.30, 0.35, 0.15, 0.20)$ to balance throughput, accepted progress, and waste. APEX-E (Efficient) uses $(0.40, 0.50, 0.10, 0.25)$ and assigns a larger penalty to waste. These weights specify training preferences; they do not guarantee that every profile is Pareto-efficient or that APEX-E always has the lowest waste. Additional ablations are evaluated separately and detailed in Appendix A.8.

5.2 Training Objective and Online Deployment

The controller uses the multi-task objective (equation 9). The survival loss predicts accepted-token survival; the cost and throughput losses calibrate runtime; the utility loss predicts scalar utility; and the ranking loss encourages the best observed depth within each training group to receive the highest predicted utility, directly matching the deployed decision rule. Appendix A.7 provides full details.

$$\mathcal{L} = \lambda_s \mathcal{L}_{\text{surv}} + \lambda_c \mathcal{L}_{\text{cost}} + \lambda_v \mathcal{L}_{\text{tps}} + \lambda_u \mathcal{L}_{\text{util}} + \lambda_r \mathcal{L}_{\text{rank}}. \tag{9}$$

At inference, the router selects $m^\star$ and $k_0^\star$. After each verifier block, APEX updates its causal state with observed acceptance, rejection behavior, and runtime statistics, scores every depth in $\mathcal{K}$, and selects the next block's depth. APEX preserves correctness while adapting to each request as it changes only the speculation depth. Algorithm 1 summarizes this procedure.

We evaluate whether APEX can maintain substantial inference acceleration while reducing discarded draft-token relative to fixed speculative configurations. We report wall-clock latency speedup and waste jointly across workloads, controller profiles, and candidate-depth sets.

Target model and serving system. We integrate APEX into vLLM with Qwen3-8B as the target verifier. All methods share the same target model; performance differences therefore arise from the proposal mechanism, speculation depth, and controller decisions. Although our evaluation uses Qwen3-8B, APEX is designed to apply broadly to autoregressive language models with compatible speculative decoding mechanisms.

Table 2: **Workload-grouped decoding performance**. Comparison of selected fixed-depth baselines with APEX. Best value per workload is in **bold**. Full per-depth sweeps discussed in Appendix A.8

.

Workload	Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
Mathematical Reasoning	EAGLE-3	$k = 8$	3.22 ×	72.25%	4.46
	n-gram SD	$k = 8$	4.15 ×	60.62%	6.84
	APEX-B	adaptive	**5.16** ×	**44.00%**	**11.72**
Long-context Completion	EAGLE-3	$k = 8$	0.51 ×	99.02%	0.52
	n-gram SD	$k = 8$	**2.03** ×	61.53%	3.30
	APEX-B	adaptive	1.83 ×	**44.42%**	**4.12**
Long-chain Reasoning	EAGLE-3	$k = 8$	1.64 ×	92.00%	1.78
	n-gram SD	$k = 8$	**3.57** ×	67.88%	5.26
	APEX-B	adaptive	3.07 ×	**40.59%**	**7.56**
Conversational Generation	EAGLE-3	$k = 8$	2.61 ×	79.88%	3.27
	n-gram SD	$k = 8$	**4.63** ×	54.63%	8.48
	APEX-B	adaptive	4.15 ×	**46.56%**	**8.91**
Code Generation	EAGLE-3	$k = 8$	2.01 ×	87.38%	2.30
	n-gram SD	$k = 8$	**3.47** ×	69.13%	**5.02**
	APEX-B	adaptive	1.99 ×	**56.83%**	3.50
SWE-style Tasks	EAGLE-3	$k = 8$	0.98 ×	99.39%	0.99
	n-gram SD	$k = 8$	**2.62** ×	76.53%	3.42
	APEX-B	adaptive	2.54 ×	**51.21%**	**4.96**

Workloads and Datasets. We evaluate six workloads: code generation using MBPP (Austin et al., 2021) and HumanEval (Chen & et al., 2021); conversational generation using UltraChat-200k (Ding et al., 2023); long-chain reasoning using GPQA (Rein et al., 2023); long-context completion using The Stack Smol (Kocetkov et al., 2022); software engineering using SWE-bench Lite (Jimenez et al., 2024); and mathematical reasoning using GSM8K (Cobbe et al., 2021) and AIME-style data (Mathematical Association of America, 1983). We use the train/test split provided by the dataset.

Baselines. We compare APEX with autoregressive decoding and fixed-depth n-gram, draft-model, and EAGLE-3 speculation at $k \in \{1, 2, 4, 8, 16\}$. These baselines expose the limitation of static depth: it can be too conservative in predictable regions and too aggressive under uncertainty.

Metrics. We use wall-clock latency speedup as the primary speed metric, values above one indicate lower end-to-end latency than autoregressive decoding: $\text{Speedup} = \frac{\text{Latency}_{\text{AR}}}{\text{Latency}_{\text{method}}}$. We also report wasted draft-token percentage: $\text{Waste}_{\%} = \frac{\text{Rejected Draft Tokens}}{\text{Total Draft Tokens}} \times 100$.

To summarize the joint speed–waste tradeoff, we introduce WAS–*Waste-Amortized Speedup* in equation 10. WAS measures speedup per unit of speculative waste, favoring methods that accelerate decoding while discarding fewer draft tokens. Higher WAS is better, and a method is Pareto-efficient if no alternative is both faster and less wasteful.

$$\text{WAS} = \frac{\text{Speedup}}{\text{Waste}_{\%}/100} = \frac{\text{Latency}_{\text{AR}}/\text{Latency}_{\text{method}}}{\text{Rejected Draft Tokens}/\text{Total Draft Tokens}} \tag{10}$$

5.3 Overall Results

We report quantitative results in Table 1, and Figure 4 highlights the speed–waste tradeoff.Increasing the fixed n-gram depth from $k = 8$ to $k = 16$ raises speedup from $3.59\times$ to $4.22\times$, while the waste fraction rises from 63.56% to 79.90%. Low waste alone does not guarantee acceleration as seen with Draft-SD at $k = 2$ has 27.82% waste but achieves only $0.96\times$ speedup

APEX-S achieves $4.27\times$ speedup with 55.30% waste; APEX-B trades speed ($3.27\times$) for lower waste (47.17%). These variants achieve the highest aggregate WAS values, 7.73 and 6.93, respec-

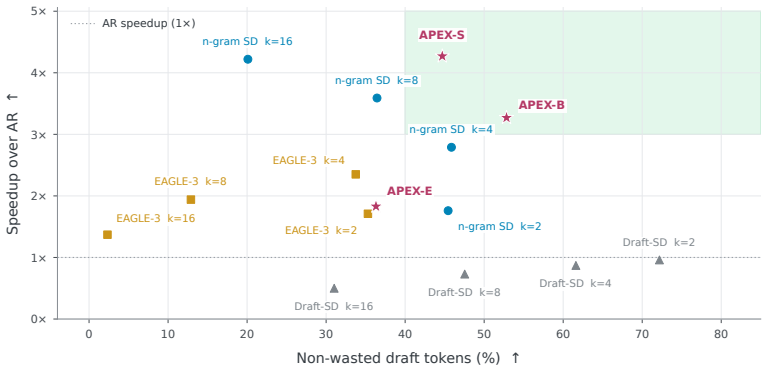


Figure 4: **Speedup vs Token Wastage.** The plot shows the trend of Speedup (vs AR) vs Token Wastage across different depths and mechanisms. The horizontal axis reports non-wasted draft tokens ($100 - \text{Waste}\%$), and the vertical axis reports speedup over autoregressive decoding.

tively. APEX-E ($1.83\times$ speedup, 63.69% waste) is dominated by both, showing that a larger training waste penalty does not guarantee lower measured waste or a better speed–waste trade-off.

5.4 Performance Across Workloads

We show workload-dependent speed–waste trade-offs in Table 2. Long-context completion and other copy-heavy regimes benefit from cheap repetition-based speculation, while reasoning, code generation, and SWE-style tasks can contain higher-uncertainty regions where early rejection makes aggressive speculation wasteful.

APEX-B improves both speed and token utilization on mathematical reasoning, achieving $5.16\times$ speedup with 44.00% waste, compared with $4.15\times$ and 60.62% for n-gram speculation at $k = 8$. Other workloads expose a trade-off rather than simultaneous improvement. On long-chain reasoning, APEX-B achieves $3.07\times$ speedup with 40.59% waste, whereas the same fixed n-gram baseline reaches $3.57\times$ with 67.88% waste. Here, adaptive control retains substantial acceleration while discarding fewer draft tokens. Tables 4 - 9 in Appendix A.8, provide more in depth study.

Collectively these results verify the motivation behind APEX-B's use of recent verifier feedback. The controller can permit deeper speculation in predictable segments and shorten drafts when acceptance becomes less reliable, accommodating variation within the same request. The benefits remain workload-dependent: on code generation, fixed n-gram speculation achieves higher speedup and WAS despite greater waste. Thus, the value of adaptation lies in balancing acceleration and token utilization, with the resulting operating point depending on the workload.

6 Conclusion and Future Work

APEX combines request-level speculator selection with block-level depth adaptation to balance decoding speed and draft-token waste. Its learned controller provides a modular way to coordinate heterogeneous proposal mechanisms while preserving target-model verification. The results demonstrate substantial acceleration with lower waste than aggressive fixed-depth configurations, with trade-offs that vary across workloads.

Future work will use traces collected during decoding to update the controller online as workloads and execution conditions change. We also plan to evaluate APEX beyond single-turn prompts, including multi-turn conversations, reinforcement-learning rollouts, and tool-augmented generation. These settings would test how well the controller responds to changes in context and generation patterns over successive steps. To expand beyond the current three speculative methods, we plan to add a common interface for new speculators and their execution traces. The router and depth controller could then be retrained on these traces to incorporate the additional methods.

7 AI USE DISCLOSURE

Generative AI tools, including OpenAI ChatGPT and Anthropic Claude, were used solely to improve the grammatical correctness, clarity, formatting, and overall flow of the manuscript and to enhance the visual quality and presentation of the methodological flow diagram. These tools were not used to formulate the research methodology, develop or modify code, design or conduct experiments, generate or analyze results, or draw scientific conclusions. All technical content, implementations, experiments, analyses, and conclusions were produced and verified by the authors. The authors reviewed all AI-assisted edits and take full responsibility for the final content of this work.

REFERENCES

Amey Agrawal et al. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Jiang Ellen, Carrie J Cai, Michael Terry, Quoc V Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Shrenik Bhansali and Larry Heck. Draft, verify, and improve: Toward training-aware speculative decoding, 2025. URL https://arxiv.org/abs/2510.05421.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024. URL https://arxiv.org/abs/2401.10774.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023. URL https://arxiv.org/abs/2302.01318.

Mark Chen and et al. Evaluating large language models trained on code, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations, 2023. URL https://arxiv.org/abs/2305.14233.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D. Lee, and Di He. Rest: Retrieval-based speculative decoding, 2024. URL https://arxiv.org/abs/2311.08252.

Yunlong Hou, Fengzhuo Zhang, Cunxiao Du, Xuan Zhang, Jiachun Pan, Tianyu Pang, Chao Du, Vincent Y. F. Tan, and Zhuoran Yang. Banditspec: Adaptive speculative decoding via bandit algorithms, 2025. URL https://arxiv.org/abs/2505.15141.

Yuxuan Hu, Ke Wang, Xiaokang Zhang, Fanjin Zhang, Cuiping Li, Hong Chen, and Jing Zhang. Sam decoding: Speculative decoding via suffix automaton, 2024. URL https://arxiv.org/abs/2411.10666.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *ArXiv*, abs/2211.15533, 2022. URL https://api.semanticscholar.org/CorpusID:254044610.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023a. URL https://arxiv.org/abs/2309.06180.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626. ACM, 2023b.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023. URL https://arxiv.org/abs/2211.17192.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test, 2025a. URL https://arxiv.org/abs/2503.01840.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty, 2025b. URL https://arxiv.org/abs/2401.15077.

Jonathan Mamou, Oren Pereg, Daniel Korat, Moshe Berchansky, Nadav Timor, Moshe Wasserblat, and Roy Schwartz. Dynamic speculation lookahead accelerates speculative decoding of large language models, 2024. URL https://arxiv.org/abs/2405.04304.

Mathematical Association of America. American invitational mathematics examination (aime), 1983. URL https://maa.org.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pp. 932–949. ACM, April 2024. doi: 10.1145/3620666.3651335. URL http://dx.doi.org/10.1145/3620666.3651335.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL https://arxiv.org/abs/2311.12022.

Dale Schuurmans, Hanjun Dai, and Francesco Zanini. Autoregressive large language models are computationally universal, 2024. URL https://arxiv.org/abs/2410.03170.

Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3909–3925, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.257. URL https://aclanthology.org/2023.findings-emnlp.257/.

Minghao Yan, Saurabh Agarwal, and Shivaram Venkataraman. Decoding speculative decoding. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6460–6473, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.328. URL https://aclanthology.org/2025.naacl-long.328/.

Mingyu Yang, Jae-Young Choi, Kihyo Moon, Minsung Jang, and Eunjoo Jeon. Dsde: Dynamic speculative decoding with kld stability for real-world serving, 2025. URL https://arxiv.org/abs/2509.01083.

Situo Zhang, Hankun Wang, Da Ma, Zichen Zhu, Lu Chen, Kunyao Lan, and Kai Yu. Adaeagle: Optimizing speculative decoding via explicit modeling of adaptive draft structures, 2024. URL https://arxiv.org/abs/2412.18910.

Xiandong Zou, Jianshu Li, Jing Huang, and Pan Zhou. Variational speculative decoding: Rethinking draft training from token likelihood to sequence acceptance, 2026. URL https://arxiv.org/abs/2602.05774.

A APPENDIX

This appendix provides the full technical details omitted from the main method section: feature construction, survival likelihood, dataset-derived regret targets, training losses, and online deployment.

A.1 BACKGROUND: AUTOREGRESSIVE DECODING

Let p_θ denote the target language model. Given a prompt or prefix $x_{\le t} = (x_1, \dots, x_t)$, autoregressive decodingSchuurmans et al. (2024) generates one token at a time according to $x_{t+1} \sim p_\theta(\cdot \mid x_{\le t})$. The newly generated token is appended to the prefix and the process repeats until a stopping condition is reached. This decoding procedure preserves the target model's generation semantics, but it introduces a strict sequential dependency: producing T output tokens requires T target-model decoding steps. For large models, each step requires loading and applying the target network to the current prefix, making decoding latency scale directly with output length.

This sequential structure is especially costly for long-form generation settings such as code synthesis, multi-step reasoning, long-context completion, and software engineering tasks. The goal of inference acceleration is therefore not merely to reduce computation, but to reduce the number or cost of sequential target-model invocations while preserving the output distribution or verification semantics of the target model.

A.2 BACKGROUND: SPECULATIVE DECODING

SD accelerates autoregressive generation by separating token proposal from target-model verification Xia et al. (2023); Leviathan et al. (2023); Chen et al. (2023). A cheaper proposal mechanism q first drafts a block of k candidate tokens, $\hat{x}_{t+1:t+k} \sim q(\cdot \mid x_{\le t})$, and the target model verifies the proposed block in parallel. If multiple draft tokens are accepted, generation advances by more than one token using a single target-model verification step.

The efficiency of this procedure depends on how many proposed tokens survive verification. Let $L_t \in \{0, \dots, k\}$ denote the accepted draft length at block t, and let r_t denote the first rejection position, with the event $r_t > k$ corresponding to full acceptance of the draft block. If rejection occurs early, all remaining draft tokens after the first rejected token are discarded. Thus, the useful progress of a speculative block is not the proposed depth k, but the accepted length L_t.

This observation makes SD a cost-sensitive procedure. A larger k increases the maximum possible progress per verifier call, but it also increases the amount of draft work that can be wasted after early rejection. The relevant quantity for efficient serving is therefore accepted progress per unit cost:

$$\mathcal{A}_t(m, k) = \frac{L_t}{C_t}, \tag{11}$$

where m denotes the speculative mechanism and C_t denotes the block-level proposal and verification cost. A speculative configuration is profitable only when the accepted progress outweighs the extra drafting and verification overhead.

A.3 PROPOSED METHOD: DETAILED STATE REPRESENTATION

The request-level state s_0 is the information available before generation begins:

$$s_0 = [e_{\text{prompt}}, \ell_{\text{prompt}}, \rho_{\text{prompt}}, \eta_{\text{prompt}}, c_{\text{decode}}] .$$

Here, e_{prompt} is a fixed-dimensional prompt embedding, ℓ_{prompt} is the prompt length, ρ_{prompt} contains repetition and lexical-structure features, η_{prompt} contains entropy or token-distribution features, and c_{decode} contains decoding configuration variables such as temperature and maximum generation length.

The block-level state s_t is the causal state available before verifier block t:

$$s_t = [e_{\text{prompt}}, \ell_{\text{prompt}}, c_{\text{decode}}, m, k, \pi_t, \mathbf{h}_{t-1}, \mathbf{g}_t] .$$

Here, m is the active speculative mechanism, k is the candidate depth being scored, π_t denotes generation progress, $\mathbf{h}_{t-1}$ summarizes verifier feedback from blocks $1, \dots, t-1$, and $\mathbf{g}_t$ contains prefix statistics computed from the generated prefix available before block t.

The history vector $\mathbf{h}_{t-1}$ includes the number of observed verifier blocks, rolling mean and variance of accepted length, first-rejection position statistics, full-acceptance rate, rejection rate, mean rejected-token count, and short-window summaries over the most recent verifier blocks. The prefix vector $\mathbf{g}_t$ includes entropy and repetition features computed over the current prefix. These features are causal: the current block outcome is never used to predict the current block.

A.4 Proposed Method: Survival Model and Accepted-Length Likelihood

The APEX-Depth network is multi-task by design. The survival head is not used merely as an auxiliary predictor; it encodes the first-rejection structure of SD.

For a candidate depth k, define J_t as the first rejected draft position using one-indexed positions. Thus, $J_t = j$ means the first $j-1$ draft tokens were accepted and the j-th draft token was rejected. If the entire block is accepted, then $J_t > k$. The accepted length is

$$L_t = \begin{cases} J_t - 1, & J_t \le k, \\ k, & J_t > k. \end{cases}$$

For each position $j \in \{1, \dots, K_{\max}\}$, the survival head predicts a discrete hazard:

$$\hat{q}_{t,j}(k) = P_\theta(J_t = j \mid J_t \ge j, s_t, m, k).$$

From these hazards, APEX computes survival probabilities and the expected accepted length. The cost head predicts the logarithm of block cost, while the throughput head predicts log throughput. These two runtime heads are necessary because acceptance alone does not determine speed: a candidate depth can accept many tokens but still be slow if proposal or verification overhead is high. The utility head predicts the final learned action value used at inference time. During deployment, APEX evaluates this utility head for every $k \in \mathcal{K}$ and selects the depth with the highest predicted utility.

The probability that the first j draft tokens survive is

$$\hat{S}_{t,j}(k) = P_\theta(J_t > j \mid s_t, m, k) = \prod_{i=1}^{j} (1 - \hat{q}_{t,i}(k)) .$$

The expected accepted length is

$$\widehat{\mathbb{E}}[L_t \mid s_t, m, k] = \sum_{j=1}^{k} \hat{S}_{t,j}(k),$$

using the identity $\mathbb{E}[L] = \sum_{j=1}^{k} P(L \ge j)$.

The likelihood of an observed accepted length is defined as follows. If $L_t = \ell < k$, then the first rejected position is $J_t = \ell + 1$, and

$$P_\theta(L_t = \ell) = \left[\prod_{j=1}^{\ell} (1 - \hat{q}_{t,j}) \right] \hat{q}_{t,\ell+1}.$$

If $L_t = k$, then the block is fully accepted and the rejection position is right-censored beyond k:

$$P_\theta(L_t = k) = \prod_{j=1}^{k} (1 - \hat{q}_{t,j}).$$

The survival loss is

$$\mathcal{L}_{\text{surv}} = -\frac{1}{N} \sum_{t=1}^{N} \log P_\theta(L_t \mid s_t, m_t, k_t),$$

where N is the number of training blocks.

A.5 Proposed Method: Dataset-Derived Regret Targets

The trace dataset provides offline expert comparisons. These quantities are used only during training. They are not available to APEX during inference.

For a request r, let $\mathcal{D}(r)$ denote the set of method-depth actions observed for that request in the offline trace dataset. We define the best observed global throughput as

$$B_{\text{global}}(r) = \max_{(m,k)\in\mathcal{D}(r)} \text{TPS}_{r,m,k}.$$

This is the best observed throughput among all speculative methods and depths for request r in the dataset.

For a fixed method m, we define the best observed same-method throughput as

$$B_{\text{method}}(r,m) = \max_{k:(m,k)\in\mathcal{D}(r)} \text{TPS}_{r,m,k}.$$

This is the best observed throughput among candidate depths for the same request and same speculative mechanism.

For a training block t belonging to request r_t, the global regret and same-method regret are

$$G_t = \frac{[B_{\text{global}}(r_t) - \text{TPS}_t]_+}{B_{\text{global}}(r_t)},$$

$$M_t = \frac{[B_{\text{method}}(r_t, m_t) - \text{TPS}_t]_+}{B_{\text{method}}(r_t, m_t)},$$

where $[x]_+ = \max(x, 0)$. The global regret G_t penalizes actions that underperform the best observed method-depth pair for the same request. The same-method regret M_t penalizes depths that underperform the best observed depth within the same speculative backend. These terms guide the controller toward actions that are competitive with strong offline alternatives.

A.6 Proposed Method: Block-Level Utility Target

For each observed training example, the block-level utility target is

$$U_t = \log(\text{TPS}_t) + \alpha_A \log(L_t + \epsilon) - \alpha_W W_t - \alpha_G G_t - \alpha_M M_t.$$

Here, TPS_t is the observed throughput, L_t is the observed accepted draft length, $W_t = R_t/k_t$ is the rejected-token rate, G_t is the dataset-derived global regret, M_t is the dataset-derived same-method regret, and $\epsilon = 0.1$ prevents numerical instability when $L_t = 0$.

The utility is designed to capture the trade-off that fixed-depth SD fails to model. High accepted length is useful only if it improves throughput, while low waste is useful only if the system still makes enough progress per verifier call. The utility target therefore combines speed, accepted progress, waste, and offline regret.

A.7 Proposed Method: Training Losses

The survival-utility controller is trained with the objective

$$\mathcal{L} = \lambda_s \mathcal{L}_{\text{surv}} + \lambda_c \mathcal{L}_{\text{cost}} + \lambda_v \mathcal{L}_{\text{tps}} + \lambda_u \mathcal{L}_{\text{util}} + \lambda_r \mathcal{L}_{\text{rank}}.$$

The cost head predicts $\widehat{\log C_t}$. Its loss is

$$\mathcal{L}_{\text{cost}} = \frac{1}{N} \sum_{t=1}^{N} \rho\left(\widehat{\log C_t} - \log C_t\right),$$

where $\rho(\cdot)$ is the SmoothL1 loss.

The throughput head predicts $\log(\widehat{1 + \text{TPS}_t})$. Its loss is

$$\mathcal{L}_{\text{tps}} = \frac{1}{N} \sum_{t=1}^{N} \rho\left(\log(\widehat{1 + \text{TPS}_t}) - \log(1 + \text{TPS}_t)\right).$$

The utility head predicts $\widehat{U}_D(s_t, m_t, k_t)$. Its regression loss is

$$\mathcal{L}_{\text{util}} = \frac{1}{N} \sum_{t=1}^{N} \rho \left(\widehat{U}_D(s_t, m_t, k_t) - U_t \right).$$

The ranking loss trains the controller to select the best depth. Candidate rows are grouped by prompt, speculative method, and block index. For a group g, let $\mathcal{K}_g$ denote the candidate depths present in that group and let

$$k_g^+ = \arg\max_{k \in \mathcal{K}_g} U_g(k)$$

be the depth with the highest observed training utility. The ranking loss is

$$\mathcal{L}_{\text{rank}} = -\frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} \log \frac{\exp(\widehat{U}_D(s_g, m_g, k_g^+))}{\sum_{k \in \mathcal{K}_g} \exp(\widehat{U}_D(s_g, m_g, k))},$$

where $\mathcal{G}$ is the set of rank groups. This term is necessary because deployment depends on the ordering of candidate depths, not only on absolute utility prediction accuracy.

In our implementation, the default loss weights are

$$(\lambda_s, \lambda_c, \lambda_v, \lambda_u, \lambda_r) = (1.00, 0.20, 0.30, 1.00, 1.00).$$

A.8 Evaluation:Extended Workload-Level Evaluation

Tables 3–9 provide aggregate and workload-level comparisons among autoregressive decoding, fixed-depth speculative baselines, and the three APEX profiles. These results expose variations hidden by aggregate averages and confirm that both the proposal mechanism and speculation depth are workload-dependent.

Aggregate performance. Table 3 shows that fixed n-gram speculation achieves a raw baseline speedup of 4.22× at k=16, but wastes 79.90% of drafted tokens. APEX-S and APEX-B achieve 4.27× and 3.27× speedup while reducing waste to 55.30% and 47.17%, respectively. Consequently, APEX-S obtains the highest overall WAS of 7.73, followed by APEX-B at 6.93, compared with 5.64 for n-gram speculation at k=8.

Sensitivity of fixed-depth baselines. The fixed baselines exhibit strong depth sensitivity. EAGLE-3 achieves its highest aggregate speedup of 2.35× at $k = 4$; increasing depth to $k = 16$ raises waste from 66.25% to 97.67% and reduces speedup to 1.37×. N-gram speculation often benefits from deeper blocks, particularly for repetitive workloads, but its waste generally increases with depth. Draft-SD rarely improves end-to-end latency despite relatively low waste at shallow depths, indicating that auxiliary proposal cost can outweigh accepted-token progress.

Workload-level behavior. APEX provides its strongest gains on mathematical reasoning and conversational generation. On mathematical reasoning, APEX-B reaches (5.16×) speedup with (44.00%) waste and a WAS of (11.72), outperforming the strongest fixed baseline in both speed and efficiency. On conversational generation, APEX-S achieves (4.88×) speedup and a WAS of (9.48), compared with (4.63×) and (8.48) for the best-speed n-gram baseline.

For long-context completion, n-gram speculation at (k=16) remains the fastest configuration at (6.40×), but APEX-S reduces waste from (66.25%) to (49.50%) while achieving a higher WAS of (10.18). A similar trade-off appears in long-chain reasoning: n-gram speculation reaches (4.57×) speedup with (77.69%) waste, whereas APEX-B achieves (3.07×) speedup with only (40.59%) waste and a higher WAS of (7.56). On SWE-style tasks, APEX-S provides (3.15×) speedup with (48.12%) waste, offering a more efficient operating point than the (4.15×), (80.31%)-waste n-gram configuration.

Failure cases and profile calibration. Code generation remains challenging for the learned controller. N-gram speculation at (k=8) and (k=16) outperforms the reported APEX profiles, while EAGLE-3 at (k=4) provides a stronger speed–waste balance than APEX-B. This result shows that

Table 3: **Overall decoding performance.** APEX is compared against autoregressive decoding and fixed speculative baselines. Higher speedup is better; lower wasted-token percentage is better.

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 2$	1.71 ×	64.71%	2.64
	$k = 4$	2.35 ×	66.25%	3.55
	$k = 8$	1.94 ×	87.11%	2.23
	$k = 16$	1.37 ×	97.67%	1.41
n-gram SD	$k = 2$	1.76 ×	54.56%	3.23
	$k = 4$	2.79 ×	54.14%	5.16
	$k = 8$	3.59 ×	63.56%	5.64
	$k = 16$	4.22 ×	79.90%	5.28
Draft-SD	$k = 2$	0.96 ×	27.82%	3.44
	$k = 4$	0.87 ×	38.39%	2.26
	$k = 8$	0.73 ×	52.46%	1.39
	$k = 16$	0.50 ×	68.98%	0.72
APEX-S	adaptive	4.27 ×	55.30%	**7.73**
APEX-B	adaptive	3.27 ×	47.17%	6.93
APEX-E	adaptive	1.83 ×	63.69%	2.87

Table 4: Workload-level speed–waste comparison for **code generation**.

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 2$	2.11×	44.50%	4.74
	$k = 4$	2.78×	55.50%	5.01
	$k = 8$	2.01×	87.38%	2.30
	$k = 16$	1.27×	98.31%	1.29
n-gram SD	$k = 2$	2.08×	46.00%	4.52
	$k = 4$	2.31×	67.25%	3.43
	$k = 8$	3.47×	69.13%	5.02
	$k = 16$	4.25×	79.69%	5.33
Draft-SD	$k = 2$	1.00×	22.06%	4.53
	$k = 4$	0.90×	33.74%	2.67
	$k = 8$	0.71×	51.88%	1.37
	$k = 16$	0.49×	69.30%	0.71
APEX-S	adaptive	2.80×	76.00%	3.68
APEX-B	adaptive	1.99×	56.83%	3.50
APEX-E	adaptive	0.99×	75.50%	1.31

adaptation does not uniformly dominate a well-matched fixed configuration and identifies code generation as an important target for improved routing and utility calibration.

APEX-E also does not consistently produce the lowest waste across workloads. APEX-B is more efficient on code, mathematics, conversational, and long-chain tasks, whereas APEX-E is more competitive on long-context and SWE-style workloads. Therefore, utility weighting alone does not guarantee a monotonic ordering of operating profiles across workload distributions; the efficiency-oriented profile may require stronger waste constraints or workload-specific calibration.

Summary. The extended results reinforce the central motivation for APEX: no proposal mechanism, fixed depth, or operating profile is universally optimal. APEX-S is generally preferable when speed is prioritized, while APEX-B provides the most consistent speed–waste balance. The remaining workload-level failures further motivate online control and improved calibration rather than reliance on a single static speculative policy.

Table 5: Workload-level speed–waste comparison for **mathematical reasoning**.

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 2$	1.55×	72.50%	2.14
	$k = 4$	2.50×	62.50%	4.00
	$k = 8$	3.22×	72.25%	4.46
	$k = 16$	2.05×	93.44%	2.19
n-gram SD	$k = 2$	1.89×	55.50%	3.41
	$k = 4$	2.85×	53.75%	5.30
	$k = 8$	4.15×	60.62%	6.84
	$k = 16$	3.74×	82.88%	4.51
Draft-SD	$k = 2$	0.99×	28.22%	3.51
	$k = 4$	0.86×	41.46%	2.07
	$k = 8$	0.66×	59.84%	1.10
	$k = 16$	0.43×	75.66%	0.57
APEX-S	adaptive	5.24×	47.00%	11.15
APEX-B	adaptive	5.16×	44.00%	11.72
APEX-E	adaptive	2.43×	72.90%	3.33

Table 6: Workload-level speed–waste comparison for **long-context completion**.

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 2$	1.59×	70.50%	2.26
	$k = 4$	1.89×	77.75%	2.43
	$k = 8$	0.51×	99.02%	0.52
	$k = 16$	1.26×	98.38%	1.28
n-gram SD	$k = 2$	1.22×	37.41%	3.26
	$k = 4$	3.32×	34.50%	9.62
	$k = 8$	2.03×	61.53%	3.30
	$k = 16$	6.40×	66.25%	9.66
Draft-SD	$k = 2$	0.89×	38.16%	2.34
	$k = 4$	1.08×	33.24%	3.24
	$k = 8$	1.07×	25.68%	4.17
	$k = 16$	0.87×	41.10%	2.12
APEX-S	adaptive	5.04×	49.50%	10.18
APEX-B	adaptive	1.83×	44.42%	4.12
APEX-E	adaptive	2.72×	41.05%	6.63

A.9 Evalutaion: Ablation Study

We evaluate the contributions of survival modeling, cost and throughput supervision, and online state features across APEX-S, APEX-B, and APEX-E. To reduce evaluation cost, Table 10 uses a diagnostic subset containing twenty samples from each of the six workloads, for a total of (120) requests. All variants are evaluated on the same requests and under the same serving configuration.

Full-controller operating profiles. The complete controllers recover the intended speed–waste trade-off. APEX-S achieves the highest speedup ((2.44×)) but incurs (59.22%) waste. APEX-E reduces waste to (52.85%) and achieves the highest WAS ((4.43)), while APEX-B provides a balanced operating point with (2.40×) speedup, (54.62%) waste, and (4.39) WAS.

Effect of survival modeling. Removing the survival loss causes the largest increase in waste across all three variants. Speedup falls to (1.62×), (1.56×), and (1.53×) for APEX-S, APEX-E, and APEX-B, respectively, while waste rises to (67.62%)–(69.38%). The corresponding WAS val-

Table 7: Workload-level speed–waste comparison for **long-chain reasoning**.

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 2$	1.46×	77.00%	1.90
	$k = 4$	2.13×	71.75%	2.97
	$k = 8$	1.64×	92.00%	1.78
	$k = 16$	1.37×	97.69%	1.40
n-gram SD	$k = 2$	1.70×	65.00%	2.62
	$k = 4$	2.46×	63.50%	3.87
	$k = 8$	3.57×	67.88%	5.26
	$k = 16$	4.57×	77.69%	5.88
Draft-SD	$k = 2$	0.98×	27.79%	3.53
	$k = 4$	0.84×	41.58%	2.02
	$k = 8$	0.68×	59.04%	1.15
	$k = 16$	0.42×	75.18%	0.56
APEX-S	adaptive	3.92×	63.50%	6.17
APEX-B	adaptive	3.07×	40.59%	7.56
APEX-E	adaptive	1.89×	58.12%	3.25

Table 8: Workload-level speed–waste comparison for **conversational generation**.

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 2$	1.78×	61.00%	2.92
	$k = 4$	2.35×	66.25%	3.55
	$k = 8$	2.61×	79.88%	3.27
	$k = 16$	1.21×	98.69%	1.23
n-gram SD	$k = 2$	1.84×	58.00%	3.17
	$k = 4$	3.05×	48.75%	6.26
	$k = 8$	4.63×	54.63%	8.48
	$k = 16$	3.20×	86.25%	3.71
Draft-SD	$k = 2$	0.91×	29.03%	3.13
	$k = 4$	0.75×	43.01%	1.74
	$k = 8$	0.58×	61.95%	0.94
	$k = 16$	0.36×	77.78%	0.46
APEX-S	adaptive	4.88×	51.50%	9.48
APEX-B	adaptive	4.15×	46.56%	8.91
APEX-E	adaptive	1.29×	74.03%	1.74

ues decrease to (2.21)–(2.37). These results show that explicitly modeling first-rejection survival is critical for selecting depths that convert drafted tokens into accepted progress.

Effect of cost and throughput supervision. Removing the cost/TPS objective causes smaller speed reductions but consistently increases waste and lowers WAS. For example, APEX-B decreases from (2.40×) speedup and (4.39) WAS to (2.33×) and (4.03), respectively. Thus, acceptance prediction alone does not capture proposal, verification, and runtime overheads.

Effect of online state features. Entropy and repetition features provide complementary content signals. Removing either increases waste across all profiles and reduces WAS, even when raw speedup changes only slightly. For APEX-S, removing entropy features increases speedup from (2.44×) to (2.49×), but also raises waste from (59.22%) to (62.41%), reducing WAS from (4.12) to (3.99). Verifier-history features are particularly important: without them, speedup falls to (1.14×)–(1.31×), and WAS decreases to approximately (2.0). This confirms that recent acceptance and rejection outcomes are essential for block-level adaptation.

Table 9: Workload-level speed–waste comparison for **swe-style** tasks.

Method	Depth	Speedup vs. AR ↑	Wasted Token % ↓	WAS ↑
EAGLE-3	$k = 2$	1.67×	66.50%	2.51
	$k = 4$	2.45×	63.75%	3.84
	$k = 8$	0.98×	99.39%	0.99
	$k = 16$	1.24×	98.50%	1.26
n-gram SD	$k = 2$	1.76×	62.00%	2.84
	$k = 4$	2.50×	62.50%	4.00
	$k = 8$	2.62×	76.53%	3.42
	$k = 16$	4.15×	80.31%	5.17
Draft-SD	$k = 2$	1.02×	20.47%	4.98
	$k = 4$	0.90×	32.70%	2.75
	$k = 8$	0.81×	46.89%	1.73
	$k = 16$	0.54×	66.06%	0.82
APEX-S	adaptive	3.15×	48.12%	6.55
APEX-B	adaptive	2.54×	51.21%	4.96
APEX-E	adaptive	2.17×	50.23%	4.32

Table 10: **Ablation study across operating profiles**. Each variant removes one component from the corresponding full APEX controller. Speedup is measured relative to autoregressive decoding AR=1.00×

Profile	Variant	Speedup	Waste %	WAS
APEX-S	**Full APEX-S**	2.44 ×	59.22 %	4.12
	w/o survival loss	1.62 ×	68.16 %	2.37
	w/o cost/TPS objective	2.41 ×	62.69%	3.85
	w/o entropy features	2.49 ×	62.41 %	3.99
	w/o repetition features	2.42 ×	62.22 %	3.89
	w/o verifier-history features	1.14 ×	54.74 %	2.09
APEX-E	**Full APEX-E**	2.34 ×	52.85 %	4.43
	w/o survival loss	1.56 ×	67.62 %	2.31
	w/o cost/TPS objective	2.26 ×	57.83 %	3.91
	w/o entropy features	2.31 ×	58.06 %	3.98
	w/o repetition features	2.29 ×	58.12 %	3.94
	w/o verifier-history features	1.31 ×	56.48 %	2.32
APEX-B	**Full APEX-B**	2.40 ×	54.62 %	4.39
	w/o survival loss	1.53 ×	69.38 %	2.21
	w/o cost/TPS objective	2.33 ×	57.83 %	4.03
	w/o entropy features	2.35 ×	58.34 %	4.03
	w/o repetition features	2.30 ×	58.45 %	3.94
	w/o verifier-history features	1.14 ×	56.22 %	2.03

Sensitivity to the candidate-depth set. We additionally evaluate all three APEX profiles under three candidate-depth interfaces (Table 11):

$$\mathcal{K} * \text{dense} = 1, \ldots, 16, \qquad \mathcal{K} * \text{deep} = 3, \ldots, 16, \qquad \mathcal{K}_{\text{sparse}} = 2, 4, 8, 12, 16. \tag{12}$$

The dense set provides maximum control flexibility, the deep set tests the importance of shallow fallback actions, and the sparse set measures sensitivity to depth granularity. Comparing speedup, waste, and WAS across these settings determines whether APEX depends on fine-grained depth choices or remains effective under a smaller deployment action space.

Table 11: APEX candidate-k sweep under the quality-balanced operating profile. Higher speedup and WAS are better; lower wasted-token percentage is better.

Method	Candidate k set	Mean k	Speedup vs. AR	Wasted Token %	WAS
APEX-S	$k = 1$–16	3.61	3.34×	47.18%	7.07
	$k = 3$–16	4.54	3.43×	50.57%	6.79
	$k \in \{2, 4, 8, 12, 16\}$	3.92	3.39×	47.95%	7.06
APEX-B	$k = 1$–16	3.09	3.25×	42.76%	7.60
	$k = 3$–16	4.28	3.34×	48.55%	6.89
	$k \in \{2, 4, 8, 12, 16\}$	3.57	3.31×	45.10%	7.33
APEX-E	$k = 1$–16	2.96	1.56×	59.88%	2.61
	$k = 3$–16	4.13	1.60×	67.83%	2.35
	$k \in \{2, 4, 8, 12, 16\}$	3.40	1.58×	62.85%	2.52