# FLOP-Efficient Training: Early Stopping Based on Test-Time Compute Awareness

**Hossam Amer[1]*, Maryam Dialameh[1,2]*, Hossein Rajabzadeh[1,2]*, Walid Ahmed[1], Weiwei Zhang[1], Yang Liu[1],**

[1]Ascend Team, Huawei Technologies, Toronto, Canada
[2]University of Waterloo, Waterloo, Canada

**Correspondence:** hossam.amer1@huawei.com

## Abstract

Scaling training compute, measured in FLOPs, has long been shown to improve the accuracy of large language models, yet training remains resource-intensive. Prior work shows that increasing test-time compute (TTC)—for example through iterative sampling—can allow smaller models to rival or surpass much larger ones at lower overall cost. We introduce TTC-aware training, where an intermediate checkpoint and a corresponding TTC configuration can together match or exceed the accuracy of a fully trained model while requiring substantially fewer training FLOPs. Building on this insight, we propose an early stopping algorithm that jointly selects a checkpoint and TTC configuration to minimize training compute without sacrificing accuracy. To make this practical, we develop an efficient TTC evaluation method that avoids exhaustive search, and we formalize a break-even bound that identifies when increased inference compute compensates for reduced training compute. Experiments demonstrate up to 92% reductions in training FLOPs while maintaining and sometimes remarkably improving accuracy. These results highlight a new perspective for balancing training and inference compute in model development, enabling faster deployment cycles and more frequent model refreshes.

## 1 Introduction

Large language models (LLMs) have driven significant progress in natural language processing and beyond (Vaswani et al., 2017; Achiam et al., 2023; Liu et al., 2024a). Open-source architectures such as Llama and Mistral demonstrate strong capabilities in text understanding and generation across diverse tasks. However, these performance gains come at a high cost: training state-of-the-art LLMs demands massive compute, large datasets, and results in considerable financial and environmental burdens (Strubell et al., 2020; Hajimolahoseini et al., 2023; Tawfilis et al., 2025). For example, training Llama-3.1-405B required 30.84 million GPU hours on NVIDIA H100 hardware, while continued pretraining or finetuning consumes over 2,000 H100 GPU hours per billion tokens. Usually, only large labs and companies have access to these resources. These challenges underscore the growing need for different strategies that improve efficiency without compromising accuracy (AI, 2024).

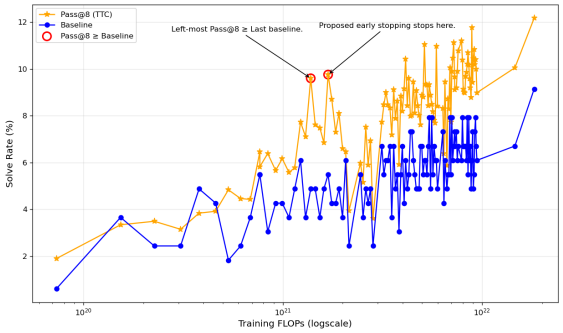



Figure 1: Solve rate (%) versus training FLOPs for TinyLlama on HumanEval. The solid blue line shows the baseline checkpoints without TTC, while the solid orange line shows checkpoints using TTC (Pass@8). The red circle indicates that TTC can achieve accuracy at least equivalent to a fully trained checkpoint without TTC, but with significantly fewer training FLOPs. Discussion and results for other $K$ values (e.g., Pass@4) are in Section 4. Based on our TTC estimation method for K, the predicted value is $K^* = 8$.

The development of LLMs has been guided by scaling laws describing how accuracy improves with increases in model size, dataset size, and compute. (Kaplan et al., 2020) established that training loss decreases as a power law with respect to model parameters, training tokens, and compute. (Hoffmann et al., 2022) later proposed the *Chinchilla paradigm*, showing that—under a fixed compute budget—optimal performance is achieved by bal-

*Equal contributions.



*Findings of the Association for Computational Linguistics: ACL 2026*, pages 35431–35449

ancing model size and training tokens, challenging the trend toward excessively large models.

While these frameworks focus primarily on training compute, recent research has begun to account for inference-time costs (Moradi et al., 2025). (Sardana et al., 2023) introduced *inference-aware scaling*, demonstrating that for high-deployment scenarios, smaller models trained longer can yield better overall cost-efficiency. Empirical studies by (Snell et al., 2024) and (Wu et al., 2024) show that increasing *test-time compute* (TTC) through methods such as verifier-guided search, or iterative sampling allow smaller models to match or even exceed the performance of much larger models at a lower total compute cost. Related work on repeated sampling and iterative refinement (Li et al., 2024; Brown et al., 2024; Chen et al., 2024) highlights the potential of multipass inference to increase accuracy without training.

Despite these advances, most studies compare fully trained models of different sizes while overlooking training dynamics within a single model, particularly how performance evolves with varying training FLOPs. This gap is critical: if TTC is considered, models could achieve competitive or even superior performance through early stopping. Existing early stopping methods, however, remain TTC-agnostic, focusing only on validation accuracy and neglecting the accuracy gains achievable at inference. Likewise, current TTC techniques, especially those based on repeated sampling, increase inference cost as $K$ grows, where $K$ denotes the number of iterative passes (Pass@$K$). These techniques do not consider predicting an optimal $K$ to run TTC more efficiently.

These shifting paradigms, from Chinchilla-optimal training to inference-aware scaling, suggest a unifying principle: modern LLM development should jointly optimize training and inference compute. We extend this principle by incorporating *TTC awareness into training*, enabling early stopping decisions based on projected TTC accuracy. Our approach reduces training FLOPs while at least preserving final accuracy, and a new small-$K$ evaluation strategy identifies the optimal $K$ with minimal inference overhead. As Figure 1 shows, TTC-aware early stopping matches and could sometimes exceed fully trained checkpoint without TTC at a fraction of the compute, *allowing faster model deployments and more frequent model refreshes*.

Overall, this paper contributes the following: 1) Reveals the *TTC-aware training* insight, an insight to reduce total FLOPs while maintaining or improving model accuracy. 2) Proposes a TTC-aware early stopping algorithm that projects TTC accuracy during training and halts when favorable FLOPs–accuracy trade-offs are achieved. This approach enables faster model refreshes and deployment cycles for large-scale model development, and also benefits academic and open-source labs with limited compute. 3) Develops an efficient TTC estimation method to predict the optimal $K$ from small-$K$ runs, lowering inference cost. 4) Provides a break-even analysis that identifies when TTC-aware training and inference match or outperform traditional training in total compute. 5) Empirically validates the effectiveness of TTC-aware training through extensive experiments.

## 2 Proposed Method: TTC-Aware Training Procedure

Figure 2 illustrates the overall proposed training procedure with TTC-awareness to achieve FLOP-efficient training. In this procedure, training hyperparameters and training FLOPs budget, $B$, are given before training. The goal is to train a model such that its final validation accuracy under TTC inference is maximized, subject to the constraint that the total training and inference costs with TTC are less than without TTC. Based on the TTC-awareness insight, we propose a FLOP-efficient training framework that matches or exceeds the validation performance of a fully trained checkpoint under budget $B$. During training, we collect validation accuracy values $\{\mathcal{A}_i\}_{i=1}^{n}$ from selected intermediate checkpoints $\{t_i\}_{i=1}^{n}$, and use them to construct a curve-fitted projection of the expected validation accuracy $\hat{\mathcal{A}}(B)$ at budget $B$. As training progresses, this projection is dynamically refined as more checkpoints are observed.

This final accuracy projection relies on empirical observations that typical learning curves follow a saturating pattern: a rapid initial increase in accuracy followed by diminishing returns (Kaplan et al., 2020). Thus, we fit the collected validation accuracy data to an exponential saturation function:

$$\hat{\mathcal{A}}(B) = a\left(1 - e^{-bB}\right) + c \tag{1}$$

where $a$ controls the scale of the gain in accuracy, $b$ determines the rate of convergence (how fast the model saturates), and $c$ represents the base accuracy at initialization. These parameters are learned

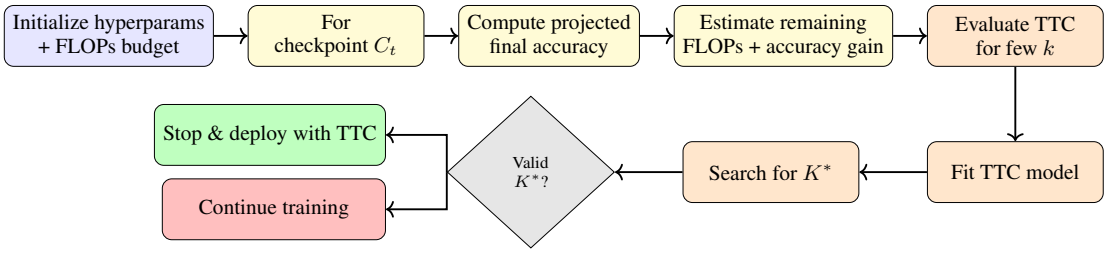



Figure 2: Flowchart of the proposed TTC-aware training procedure. The method efficiently selects the intermediate checkpoint with TTC configuration that is superior in terms of FLOPs and accuracy than the fully trained checkpoint, enabling early stopping. Please see Algorithm 1 and the associated explanation in Appendix F.

by minimizing the mean squared error between observed validation accuracies and the fitted curve. The fitted function $\hat{\mathcal{A}}(B)$ is then used to predict final validation performance under budget $B$, guiding early stopping and inference allocation.

During training, validation accuracy typically exhibits fluctuations, which complicates fitting $\hat{\mathcal{A}}(B)$. To mitigate the noisy impact of such fluctuations, we retain only validation points that are monotonically non-decreasing with respect to the best accuracy observed so far; specifically, a point is included if its validation accuracy is greater than or equal to the maximum previously observed value, and discarded otherwise. More generally, this criterion can be relaxed by introducing a user-defined tolerance around the running maximum; however, for simplicity and to avoid introducing additional hyperparameters, we adopt the strict $\geq$ rule in all experiments. We then fit the model exclusively to this filtered, rising portion of the curve. Consequently, the estimator does not introduce additional variance beyond that inherent to the underlying training run. This design choice reflects our objective of accurately estimating the maximum achievable accuracy used to make early stopping decisions (see Figures 7 and 8).

To support this framework with efficient inference, we introduce a TTC-aware mechanism that estimates the optimal number of inference samples per query, denoted $K^*$, without incurring excessive compute. Since standard TTC performs inference by sampling the model $K$ times and aggregating the responses, it can become prohibitively expensive if $K$ is large. Instead, we propose to approximate the relationship between validation accuracy under TTC at time $t$ and the number of samples $K$ using a sigmoid curve:

$$\hat{\mathcal{A}}_K(t) = \frac{L}{1 + e^{-k(K - x_0)}}, \tag{2}$$

where $L$ is the maximum achievable TTC accuracy, $k$ controls the growth rate of accuracy with increasing $K$, and $x_0$ is the inflection point, where accuracy reaches half of its maximum. This relation aligns with the intuitive expectation that increasing $K$ will eventually lead to a saturation point, beyond which accuracy gains level off (more evidence is in Figure 9 in the appendix).

These parameters are estimated using non-linear least squares fitting over a few sampled values of $K$, obtained by querying the model at small $K$ values (e.g., $K = 1, 2, 4$). Evaluating once at $K$ implicitly yields predictions for all $K' \leq K$, so querying at $K = 4$ provides the effective results for $K = 1, 2, 4$ at no additional cost. This formula allows us to extrapolate and predict $K^*$—the smallest value of $K$ that meets both budget and accuracy constraints. The optimal $K^*$ must satisfy the total FLOPs constraint:

$$\begin{aligned} \mathrm{F}_{\text{tr}}[t] + \mathrm{F}_{\text{inf}}[t, K^*] < \\ \mathrm{F}_{\text{tr}}[B] + \mathrm{F}_{\text{inf}}[B, 1] \end{aligned} \tag{3}$$

where $\mathrm{F}_{\text{tr}}[t]$ is the training cost up to timestep $t$, $\mathrm{F}_{\text{inf}}[t, K^*]$ is the cost of evaluating TTC inference at that timestep using $K^*$ samples. In addition, $K^*$ must meet the projected accuracy requirement:

$$\hat{\mathcal{A}}_{K^*}(t) \geq \hat{\mathcal{A}}(B) \tag{4}$$

That is, the TTC accuracy at timestep $t$ with $K^*$ samples must be at least as good as the projected final validation accuracy under full budget $B$. If such a $K^*$ exists after a patience window of $p$ checkpoints, training is terminated early; otherwise, training continues to the next checkpoint.

## 3 Early Stopping Computational Overhead and Break-Even Bound

The proposed TTC-aware training procedure with early stopping can introduce additional computa-

**Algorithm 1** TTC-Aware Exponential-Fit Early Stopping

**Input:** Patience $p$, training budget $B$; **Init:** $best\_val\_acc, best\_ttc\_val\_acc \leftarrow -\infty, patience\_counter \leftarrow 0$
Define fitting function $f(x) = a(1 - e^{-bx}) + c$
**for** FLOPs $t = 3,4, \ldots$ **until** $t > B$ **do**
  Train one step; evaluate to obtain $val\_acc_t$
  Fit $f(x)$ using observed FLOPs and $\{val\_acc\}$; estimate baseline final accuracy $f(B)$
  Find **minimum** $K^*$ satisfying: $F_{\mathrm{tr}}(t) + F_{\mathrm{inf}}(t, K^*) < F_{\mathrm{tr}}(B) + F_{\mathrm{inf}}(B, 1)$ & $\mathrm{Acc}_t^{\mathrm{ttc}}(K^*) \geq f(B)$
  Update $patience\_counter \leftarrow 0$ if $val\_ttc\_acc_t[K^*] > best\_ttc\_val\_acc$
  $best\_ttc\_val\_acc \leftarrow \max(best\_ttc\_val\_acc, val\_ttc\_acc_t[K^*])$
  Update $patience\_counter \leftarrow 0$ if $best\_ttc\_val\_acc < f(B)$ else $patience\_counter{+}{+}$
  **if** $patience\_counter \geq p$ **then**
    **Stop**; load best TTC checkpoint; **break**
  **end if**
**end for**

tion, because TTC-driven inference is periodically performed both during training and deployment. It is therefore useful to determine *when* these extra costs are offset by the training savings from TTC-aware early stopping. We refer to this as the *break-even* point: the largest amount of downstream inference that can be performed before the total cost of "TTC-aware training + deployment" exceeds the cost of training the same model *without* TTC.

Here, the total cost explicitly includes both the training cost (which may already incorporate TTC-driven inference) and the deployment cost, i.e., the FLOPs or tokens required to serve end-user queries. Since modern LLMs are not trained once and frozen indefinitely, but rather are *continuously refreshed* through pretraining, finetuning, and ongoing updates (Ramaswamy et al., 2019; Gamal et al., 2023; Amer et al., 2024). Thus, characterizing this bound is especially important: it indicates the consumption of inference tokens before the model should be refreshed.

We denote the number of *training* tokens for the baseline schedule (without TTC) by $N_{\text{train}}$, and the number of *inference* tokens $N_{\text{infer}}$. Here, $N_{\text{infer}}$ accounts for all inference tokens, including those used during TTC sampling as well as tokens served to end users. The ratio of training FLOPs with TTC relative to the baseline is $r \in (0, 1]$, where $r < 1$ indicates training savings from earlier stopping. The inference cost of serving the model with TTC is captured by a multiplier $\lambda > 1$, relative to serving the baseline model without TTC. Assuming that training FLOPs per token are roughly $6\times$ the inference FLOPs per token (accounting for forward + backward passes and optimizer updates), the break-even condition — i.e., the maximum number of inference tokens that can be served while ensuring TTC does not exceed the total FLOPs of the baseline without TTC — is:

$$N_{\text{infer}} \leq \frac{6\,(1-r)}{\lambda - 1} \cdot N_{\text{train}}. \tag{5}$$

A detailed derivation and discussion are provided in Appendix A.

For illustrative purposes, Table 1 reports the number of inference samples permitted under our bound, given a model output sequence length of 1024 and a total of 3T training tokens. When $\lambda \approx 1.2\times$, inference with TTC can be applied for up to 70B samples before requiring the next model refresh. This also motivates the importance of the design of even more efficient TTC inference techniques, which we add to future work.

| r | $\lambda$ | $N_{\text{infer}}$ | Inference Samples |
|---|---|---|---|
| 0.4 | 8 | 1.54T | 1.5B |
| 0.2 | 16 | 0.96T | 937M |
| 0.4 | 1.2 | 54T | 52.7B |
| 0.2 | 1.2 | 72T | 70.3B |

Table 1: Illustrative values for $r$, $\lambda$, $N_{\text{infer}}$, and inference samples where each sample has 1024 tokens.

## 4 Experimental Results

### 4.1 Experimental Setup

**Models and Checkpoints** We evaluate our approach using three open-source model families: TinyLlama (Zhang et al., 2024), Pythia (Biderman et al., 2023), FineMath (Liu et al., 2024b), and Qwen3 (Yang et al., 2025). These models were selected because they provide publicly available intermediate checkpoints, unlike most large-scale models where reproducing training would be prohibitively expensive. Table 2 summarizes the models under evaluation. We use TinyLlama and Pythia to assess the impact of TTC-aware training in a *pre-training* setting, while FineMath enables evaluation in a *continued training* scenario.

**Datasets** We evaluate our models on a diverse set of benchmarks spanning code generation, reading comprehension, and mathematical reasoning.

| | TinyLlama | Pythia | FineMath | Qwen3-A3B-Instruct |
|---|---|---|---|---|
| Size (B) | 1.1 | 1 / 2.8 / 6.9 | 3 | 30 |
| Arch. Base | LLaMA-2 (GQA) | GPT-style | LLaMA-3 | Qwen3 |
| Training Data | SlimPajama StarCoder | The Pile | FineWeb-Edu FineMath | Open Data |
| Tokens | ∼2T | 300B | 160B | 36T |
| Optimizer | AdamW | Adam + ZeRO | AdamW | – |

Table 2: Models used in our experiments.

Specifically, we include HumanEval (Li et al., 2022), a standard benchmark for assessing code synthesis from natural language prompts; DROP (Dua et al., 2019), which tests discrete reasoning and reading comprehension over passages; Math-500 (Hendrycks et al., 2021), a curated set of advanced mathematics problems; GSM8k (Cobbe et al., 2021), a collection of grade-school math word problems designed to evaluate multi-step arithmetic reasoning.

**Inference Hardware and Environment** Inference experiments are carried out on a 8xNVIDIA Tesla V100 GPU. Float16 inference is enabled because it speeds up the inference while maintaining accuracy. Our inference environment is built on top of the InstructEval benchmark (Chia et al., 2023).

**TTC Hyperparameter Settings** Our experiments use a temperature of 0.8, a patience value of 10, a maximum input sequence length of 1,024 tokens, and a maximum output length of 512 tokens. The rest of configurations are borrowed from InstructEval repository (Chia et al., 2023).

### 4.2 TTC-Aware Training Insight

TTC-Aware training allows models to reach equal or better accuracy using significantly fewer training FLOPs or less data. This section presents experiments to demonstrate this insight.

**TinyLlama on HumanEval and DROP:** As shown in Figure 1, an intermediate checkpoint achieves at least the same or even better accuracy than the fully trained baseline without TTC, while providing up to 92.44% training FLOP savings. In particular, this checkpoint produces a positive accuracy gain of 0.45%. When accounting for the additional FLOPs consumed by TTC inference during training, the total savings remain substantial at 85.5%. Although the final checkpoint with Pass@8 attains higher accuracy, this improvement comes at the cost of a substantially larger FLOPs budget compared to the intermediate checkpoint. To further validate this insight, Figure 10 exhibits the same trend on DROP, where an earlier checkpoint achieves comparable or better accuracy under a much smaller training budget. This confirms that the TTC-aware training phenomenon is not limited to a single dataset, but rather generalizes across tasks.

**TinyLlama on HumanEval across different Pass@K metrics:** In Figure 3, we observe three consistent trends. First, there always exists an intermediate checkpoint where the Pass@K performance is at least as strong as that of the fully trained baseline without TTC, while requiring substantially fewer FLOPs. Second, when comparing across different $K$ values, if the intermediate checkpoint uses a *larger* $K$ than the fully trained model (e.g., Pass@8 or Pass@16 vs. Pass@4), it can often maintain or even surpass the accuracy of the fully trained baseline while still providing FLOPs savings. Under TTC, if the first checkpoint used a smaller $K$ than the fully trained, it attains lower accuracy. To reach comparable performance, training must proceed to the full budget. Third, when both models are compared at the same $K$, the fully trained baseline attains higher accuracy, but only at the cost of a substantially larger FLOPs budget.

**Pythia on HumanEval, GSM8K:** For each Pythia 1B/2.8B/6.9B model, we compare an intermediate checkpoint at 80k training steps with the fully trained baseline at 143k training steps (see Table 3). Across both HumanEval and GSM8k, we observe the same TTC-aware trends identified earlier with TinyLlama. Intermediate checkpoints with TTC match or exceed the performance of fully trained baselines while requiring substantially fewer FLOPs. For example, on HumanEval, the 1B Pythia model at 80k achieves Pass@8 accuracy (3.8%) comparable to its 143k baseline without TTC (3.04%), while the 2.8B model at 80k with Pass@8 (8.84%) not only surpasses its 143k baseline (6.7%) but also outperforms the much larger 6.9B baseline at 143k (6.8%).

A similar pattern appears on GSM8k: the 2.8B checkpoint at 80k matches or exceeds the 143k baseline across all Pass@K values, and the 6.9B checkpoint at 80k with TTC outperforms its fully trained counterpart without TTC. Furthermore, when comparing different $K$ values within the same model size, if the intermediate checkpoint uses a larger $K$ than the fully trained baseline, its accuracy is often higher. For example, Pythia-2.8B at 80k with Pass@8 (11.29%) surpasses Pythia-2.8B at 143k with Pass@4 (6.3%). Consistent patterns are also observed across Pythia model sizes,

| Model | Dataset | Step | Tokens | FLOPs | Baseline | Pass@4 | Pass@8 | Pass@16 | Pass@32 | Pass@64 |
|---|---|---|---|---|---|---|---|---|---|---|
| Pythia-1B | HumanEval | 80,000 | 167.8T | $9.6\times10^{20}$ | 1.8 | 2.8 | **3.8** | 4.9 | 6.4 | 7.9 |
| Pythia-1B | HumanEval | 143,000 | 299.9T | $1.7\times10^{21}$ | **3.04** | 4.01 | 5.5 | 7.5 | 9.8 | 12.2 |
| Pythia-1B | GSM8k | 80,000 | 167.8T | $9.6\times10^{20}$ | 1.74 | 6.06 | 10.99 | 18.07 | 27.17 | 35.24 |
| Pythia-1B | GSM8k | 143,000 | 299.9T | $1.7\times10^{21}$ | 2.20 | 6.97 | 11.22 | 18.34 | 27.52 | 36.16 |
| Pythia-2.8B | HumanEval | 80,000 | 167.8T | $1.93\times10^{21}$ | 8.5 | 6.47 | **8.84** | 11.49 | 14.17 | 17.07 |
| Pythia-2.8B | HumanEval | 143,000 | 299.9T | $3.43\times10^{21}$ | **6.7** | 8.38 | 11.02 | 14.01 | 17.53 | 20.73 |
| Pythia-2.8B | GSM8k | 80,000 | 167.8T | $1.93\times10^{21}$ | 1.54 | 6.29 | 11.29 | 18.19 | 26.99 | 35.86 |
| Pythia-2.8B | GSM8k | 143,000 | 299.9T | $3.43\times10^{21}$ | 2.04 | 6.30 | 11.30 | 18.73 | 27.75 | 38.21 |
| Pythia-6.9B | HumanEval | 80,000 | 167.8T | $6.7\times10^{21}$ | 6.8 | 8.17 | 11.09 | 13.96 | 16.98 | 20.12 |
| Pythia-6.9B | HumanEval | 143,000 | 299.9T | $1.2\times10^{22}$ | 6.8 | 10.28 | 13.22 | 16.40 | 20.15 | 24.39 |
| Pythia-6.9B | GSM8k | 80,000 | 167.8T | $6.7\times10^{21}$ | 1.74 | 7.05 | 12.28 | 18.50 | 29.04 | 40.33 |
| Pythia-6.9B | GSM8k | 143,000 | 299.9T | $1.2\times10^{22}$ | 2.80 | 7.53 | 13.16 | 19.27 | 29.67 | 40.79 |

Table 3: Comparison of intermediate (80k) and final (143k) checkpoints across Pythia models of different sizes on HumanEval and GSM8K. Results are reported using pass@k metrics. **Bold** entries indicate cases where an intermediate TTC-aware checkpoint matches or surpasses the fully trained baseline at the same or higher $K$, highlighting the consistency and benefit of TTC-aware training.

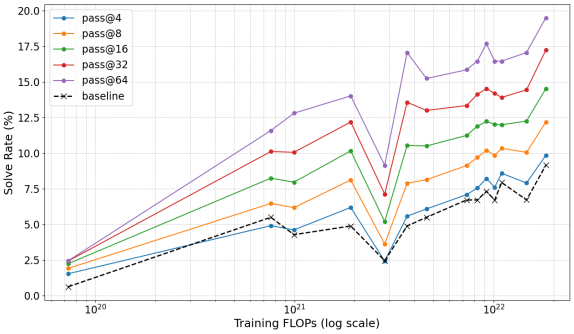



Figure 3: TinyLlama on HumanEval across Pass@$K$. Intermediate checkpoints can match or exceed the fully trained baseline with far fewer FLOPs, sometimes outperforming the baseline at higher $K$, while the final checkpoint is better only at the cost of much larger FLOPs.

reinforcing the findings from TinyLlama: the TTC-aware effect generalizes across model scales and datasets, demonstrating that intermediate checkpoints can achieve comparable or better performance at a fraction of the training cost.

**FineMath with actual TTC techniques:** In all previous experiments, we focused on Pass@$K$, which captures the potential performance of a model under multiple sampled outputs. To extend this to more advanced TTC techniques beyond Pass@$K$, we evaluate several approaches: (1) the Baseline without TTC, (2) Pass@$K$, (3) Majority Voting, (4) Diverse Tree Search (DVTS) with naive weighting, (5) DVTS with weighted aggregation, and (6) Compute-Optimal Search, which selects the best-performing method among Majority Voting and the two DVTS variants (Beeching et al., 2024; Snell et al., 2024).

In DVTS with weighted aggregation, identical responses are grouped, and the final answer is chosen based on the highest cumulative reward across all identical responses. In contrast, DVTS with naive weighting treats each generated response independently, selecting the one with the highest reward model (RM) score as the final output.

We apply these techniques to the FineMath model in a continued-training setup at 10B, 30B, 40B, and 50B training tokens within the mathematics domain and evaluate on the Math-500 benchmark. An important observation from Table 4 is that even when a lower training budget underperforms at a given Pass@$K$, there often exists a larger $K$ where it surpasses higher-budget checkpoints evaluated at smaller $K$. For example, at $K$=2 the 10B checkpoint underperforms the 50B one (15.05% vs. 18.28%), but at $K$=4, the 10B checkpoint outperforms the 50B checkpoint with $K$=2 (24.73% vs. 18.28%). This pattern recurs across multiple settings (e.g., 10B vs. 40B), illustrating the broader TTC-aware insight in a continued training scenario as shown with FineMath.

We further find that this TTC-aware insight generalizes beyond Pass@$K$ to verifier-based and aggregation metrics. For instance, the 10B checkpoint with $K$=16 under the DVTS-WEIGHTED verifier achieves a 19.35% solve rate, surpassing the 50B checkpoint without TTC (15.05%). Similarly, the 50B checkpoint improves from 13.98% at $K$=2 to 21.51% at $K$=4, demonstrating the consistent performance gains achieved with larger $K$. To reach comparable or better accuracy than the 50B checkpoint, one could instead use the 10B checkpoint with $K$=32 (33.33%) or the 30B checkpoint with $K$=16 (24.73%) or the 40B checkpoint with $K$=32 (41.94%). Across all TTC techniques evaluated, we observe that solve rates generally in-

crease as $K$ grows. However, verifier-based TTC scores are typically lower than raw Pass@$K$ values, underscoring the importance of designing strong verifiers for realistic evaluation. Among the TTC techniques, DVTS-WEIGHTED consistently yields the best overall performance, while the Compute-Optimal Search provides an approximate upper bound over the tested methods. These findings reinforce that the TTC-aware insight persists even under verifier-based and aggregation metrics, further validating the generality of the TTC framework across both metric types and model scales.

**TinyLlama on Math-500** We evaluate TinyLlama on the Math-500 benchmark, a standard dataset for mathematical reasoning tasks. Math-500 is particularly challenging for TinyLlama because the model was not originally trained on sufficient math-domain data, resulting in a 0% score for the original fully trained checkpoint. To address this, we conducted additional training to include enough math data, despite our limited compute resources. Table 9 details the full training configuration.

Due to these compute constraints, we were only able to use a subset of intermediate checkpoints. Nevertheless, even with this partial coverage, the baseline without TTC shows a roughly monotonic increase across available checkpoints, albeit with some variation in scores. As shown in Figure 4, the results reveal two clusters of points: one corresponding to checkpoints trained from scratch on cosmopedia dataset with varying amounts of FLOPs appearing on the left, and another corresponding to fine-tuned checkpoints with varying amount of FLOPs appearing on the right.

Despite these practical limitations, the TTC-aware insight remains visible. This demonstrates both the promise of TTC for improving training efficiency and the challenges in fully exploiting it across models and checkpoints, highlighting the need for appropriate training resources and setup in future work.

**Results on Large Frontier Models:** While evaluating our method on frontier-scale LLMs is limited to the availability of intermediate checkpoints, we created the following setting: we fine-tuned Qwen3-30B-A3B-Instruct for 1 epoch on GSM8K and saved intermediate checkpoints throughout training and applied our TTC selection over them. As shown in Figure 5, even in this large-model setting, we observe the same TTC-aware effect: the early TTC-selected intermediate checkpoint with Pass@2 can outperform the final checkpoint by more than 10%, demonstrating that TTC remains beneficial beyond the small-model regime.

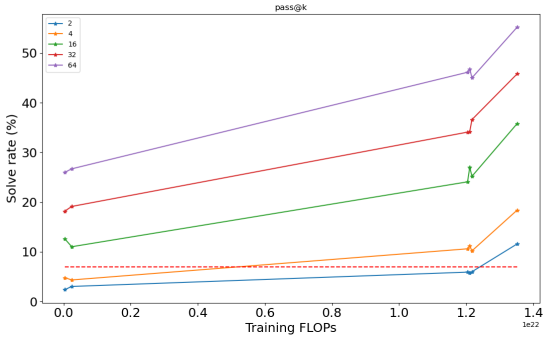



Figure 4: TinyLlama on Math-500 dataset with different Pass@K. The dotted horizontal line represents the score of the fully trained TinyLlama without TTC. Challenging setup given the TinyLlama available checkpoints accuracy on Math-500 and our compute resources.

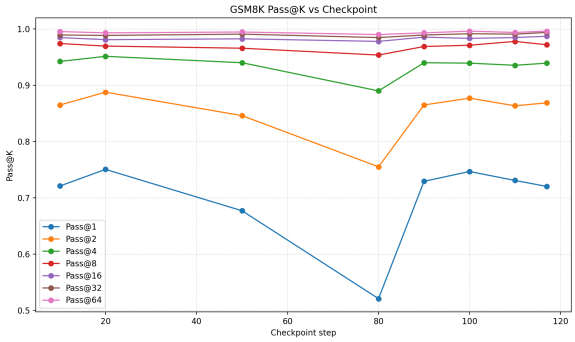



Figure 5: Qwen3-30B-A3B-Instruct on GSM8K (1 epoch). Pass@K over intermediate checkpoints shows non-monotonic behavior; TTC selects an intermediate checkpoint that outperforms the final checkpoint.

### 4.3 Early Stopping Analysis and Results

The proposed early stopping algorithm achieves noticeable gains as given in Table 5. In TinyLlama with the HumanEval dataset, it reaches Pass@8 accuracy roughly 0.6% higher than the fully trained checkpoint without TTC while providing 90.7% training FLOP savings, and delivers a 4.3% improvement in Pass@16 with 90.76% training FLOP savings. Using our early stopping method, Pass@8 reduces training FLOPs by 90.7% relative to the baseline, while we could practically reach 92.44% overall FLOP savings. Similar trends hold across datasets and model families, including Pythia (Biderman et al., 2023). Overall, TTC-aware training can save up to 92% of training FLOPs without

| Train Tokens | $K$ | Baseline | Pass@K | Majority | Naive | Weighted | CTS |
|---|---|---|---|---|---|---|---|
| 10B | 2 | 12.90 | 15.05 | 12.90 | 12.90 | 12.90 | 13.98 |
| | 4 | 12.90 | 24.73 | 11.83 | 11.83 | 12.90 | 16.13 |
| | 16 | 12.90 | 53.76 | 19.35 | 21.51 | 19.35 | 29.03 |
| | 32 | 12.90 | 67.74 | 26.88 | 23.66 | 33.33 | 44.09 |
| | 64 | 12.90 | 74.19 | 25.81 | 24.73 | 34.41 | 39.78 |
| 30B | 2 | 7.53 | 12.90 | 8.60 | 11.83 | 11.83 | 12.90 |
| | 4 | 7.53 | 21.51 | 11.83 | 19.35 | 20.43 | 21.51 |
| | 16 | 7.53 | 46.24 | 19.35 | 24.73 | 24.73 | 33.33 |
| | 32 | 7.53 | 61.29 | 23.66 | 23.66 | 27.96 | 36.56 |
| | 64 | 7.53 | 68.82 | 31.18 | 26.88 | 35.48 | 45.16 |
| 40B | 2 | 11.83 | 13.98 | 11.83 | 12.90 | 12.90 | 13.98 |
| | 4 | 11.83 | 24.73 | 15.05 | 18.28 | 17.20 | 21.51 |
| | 16 | 11.83 | 51.61 | 29.03 | 22.58 | 29.03 | 38.71 |
| | 32 | 11.83 | 67.74 | 37.63 | 27.96 | 41.94 | 50.54 |
| | 64 | 11.83 | 73.12 | 39.78 | 32.26 | 49.46 | 52.69 |
| 50B | 2 | 15.05 | 18.28 | 13.98 | 13.98 | 13.98 | 15.05 |
| | 4 | 15.05 | 29.03 | 18.28 | 18.28 | 21.51 | 24.73 |
| | 16 | 15.05 | 59.14 | 36.56 | 27.96 | 37.63 | 45.16 |
| | 32 | 15.05 | 66.67 | 38.71 | 32.26 | 39.78 | 49.46 |
| | 64 | 15.05 | 76.34 | 40.86 | 34.41 | 44.09 | 53.76 |

Table 4: Solve Rate (%) of *FineMath* checkpoints (10B, 30B, 40B, 50B tokens) across different $K$. The Verifier is Skywork-o1-Open-PRM-Qwen-2.5-1.5B. We report Baseline without TTC, Pass@$k$, Majority, DVTS Naive, DVTS Weighted, and Compute-Optimal Search (CTS) aggregation methods on Math-500.

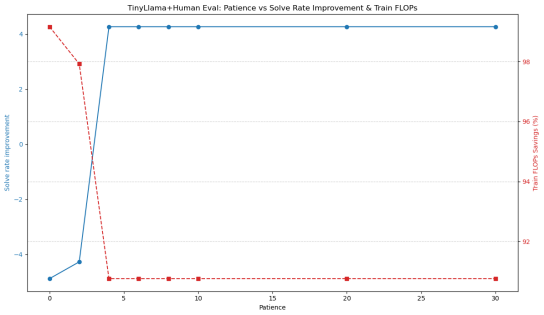



Figure 6: Effect of patience on TinyLlama (HumanEval). Higher patience improves TTC Pass@16 but increases training FLOPs; a mid-range value ($\approx$ 10) offers the best trade-off, while overly large patience reduces early-stopping benefits.

sacrificing accuracy and often improves it.

To further benchmark the proposed early stopping algorithm, we implement PyTorch-style naive early stopping and evaluate it on TinyLlama–HumanEval (PyTorch-Ignite, 2025). We vary the patience from 0 to 100 with $\epsilon = 0$. We report $\mathbf{\Delta}$, the accuracy difference between the early-stopped model and the fully trained baseline; FLOPs saving, the fraction of training compute saved; and $\mathbf{\Delta}_{\text{TTC}}$, the accuracy difference after applying TTC Pass@8 to the early-stopped checkpoint. As shown in Table 6, naive early stopping exhibits a poor trade-off: it either halts very early, yielding large FLOPs savings but substantial accuracy degradation, or stops near full training with negligible savings. Applying TTC post hoc does not meaningfully improve this trade-off. Overall, naive early stopping does not perform as the proposed method, which can achieve up to 92% FLOPs savings while improving accuracy by 0.44%.

Table 5: Effect of validation fluctuation filtering on early stopping for TinyLlama.

| | HumanEval | | DROP | |
|---|---|---|---|---|
| **Method** | **FLOPs Saving** | **Acc. Gain** | **FLOPs Saving** | **Acc. Gain** |
| Early stopping w/ fluctuations | 56.8% | +2.05% | 72.7% | +2.08% |
| Early stopping w/o fluctuations | 90.7% | +0.6% | 87.0% | +1.0% |

Table 6: Relative accuracy and computational cost (FLOPs) of naive early stopping versus the fully trained TinyLlama checkpoint on HumanEval.

| **Patience** | $\mathbf{\Delta}$ (%) | **FLOPs Saving** (%) | $\mathbf{\Delta}_{\text{TTC}}$ (%) |
|---|---|---|---|
| 0 | $-4.27$ | 97.92 | $-5.31$ |
| 6 | $-3.66$ | 95.80 | $-2.69$ |
| 8 | $-3.05$ | 93.28 | $-1.42$ |
| 10 | $-3.05$ | 93.28 | $-1.42$ |
| 20 | $-1.22$ | 70.24 | 0.19 |
| 50 | 0.00 | 0.00 | 3.03 |
| 100 | 0.00 | 0.00 | 3.03 |

## 5 Conclusion

We introduced *TTC-aware training*, which selects an intermediate checkpoint together with a minimal test-time compute setting $K^*$ to maximize quality per total FLOPs, rather than defaulting to the final checkpoint at budget $B$. Across multiple model scales and settings, this yields substantial savings with no quality loss: we achieve up to **92.44%** training-FLOPs reduction while preserving (and often improving) downstream performance. For example, an **80k-step** Pythia-2.8B checkpoint with TTC reaches **8.84%** Pass@8 on HumanEval versus **6.7%** for the fully trained baseline, and verifier-based TTC on Math-500 improves from **15.05%** (50B-token baseline) to **19.35%** at a **10B-token** checkpoint with $K$=16. Our TTC-aware early stopping strategy further strengthens this result: on TinyLlama–HumanEval, it improves Pass@8 by roughly **+0.6%** over the fully trained checkpoint (without TTC) while saving **90.7%** training FLOPs, and achieves **+4.3%** Pass@16 with **90.76%** FLOPs savings. Finally, our break-even bound shows that these training savings can fund TTC for $\sim$**70B** downstream samples (for $\lambda \approx 1.2\times$), making TTC-aware training especially attractive for frequently refreshed models.

## Limitations

**Models and Checkpoints.** Our study is constrained to publicly available checkpoints up to 6.9B parameters. Larger models such as Llama-3.x-405B do not release intermediate checkpoints, preventing direct evaluation.

**Datasets.** Some model–dataset combinations (e.g., TinyLlama on Math-500) perform poorly due to limited domain pretraining and compute, restricting evaluation breadth.

**TTC Inference Efficiency.** Realizing TTC gains in deployment may require efficient inference strategies (e.g., beam search, diverse search) to control latency and cost. Integrating such optimizations remains an important direction.

**Verifier-Based Evaluation.** We primarily evaluate verifier-based methods in continued pretraining. A broader study across multiple verifiers and architectures is valuable future work, but our focus here is demonstrating FLOP reductions while maintaining accuracy.

**Scaling Laws.** A full TTC-aware training scaling law—characterizing the optimal stopping point—remains open and is an important avenue for future research.

## References


Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Meta AI. 2024. Llama 3.1 405b. https://huggingface.co/meta-llama/Llama-3.1-405B. Accessed: 2025-08-21.

Hossam Amer, Joe Osborne, Michael Zaki, and Mohamed Afify. 2024. On-device emoji classifier trained with gpt-based data augmentation for a mobile keyboard. *arXiv preprint arXiv:2411.05031*.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, and 1 others. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Edward Beeching, Lewis Tunstall, and Sasha Rush. 2024. Scaling test-time compute with open models. Hugging Face Space. https://huggingface.co/spaces/HuggingFaceH4/blogpost-scaling-test-time-compute.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, and 1 others. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei A Zaharia, and James Y Zou. 2024. Are more llm calls all you need? towards the scaling properties of compound ai systems. *Advances in Neural Information Processing Systems*, 37:45767–45790.

Yew Ken Chia and 1 others. 2023. Instructeval: Towards holistic evaluation of instruction-tuned large language models. *arXiv preprint arXiv:2306.04757*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 2368–2378. Association for Computational Linguistics.

Karim Gamal, Ahmed Gaber, and Hossam Amer. 2023. Federated learning based multilingual emoji prediction in clean and attack scenarios. *arXiv preprint arXiv:2304.01005*.

Habib Hajimolahoseini, Walid Ahmed, and Yang Liu. 2023. Training acceleration of low-rank decomposed networks using sequential freezing and rank quantization. *arXiv preprint arXiv:2309.03824*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, and 1 others. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. 2024. More agents is all you need. *arXiv preprint arXiv:2402.05120*.

X Li, DG Wang, S Wang, S Wang, Y Wang, Y Wang, Y Wang, Z Wang, Z Wang, and 1 others. 2022. Evaluating large language models trained on code. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 12345–12356.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Yan Liu, Renren Jin, Ling Shi, Zheng Yao, and Deyi Xiong. 2024b. Finemath: A fine-grained mathematical evaluation benchmark for chinese large language models. *arXiv preprint arXiv:2403.07747*.

Mohammad Mahdi Moradi, Hossam Amer, Sudhir Mudur, Weiwei Zhang, Yang Liu, and Walid Ahmed. 2025. Continuous self-improvement of large language models by test-time training with verifier-driven sample selection. *arXiv preprint arXiv:2505.19475*.

PyTorch-Ignite. 2025. Earlystopping — pytorch-ignite documentation. https://docs.pytorch.org/ignite/generated/ignite.handlers.early_stopping.EarlyStopping.html. Accessed: 2025-12-16.

Swaroop Ramaswamy, Rajiv Mathews, Kanishka Rao, and Françoise Beaufays. 2019. Federated learning for emoji prediction in a mobile keyboard. *arXiv preprint arXiv:1906.04329*.

Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. 2023. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws. *arXiv preprint arXiv:2401.00448*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 13693–13696.

Youssef Tawfilis, Hossam Amer, Minar El-Aasser, and Tallal Elshabrawy. 2025. A distributed generative ai approach for heterogeneous multi-domain environments under data sharing constraints. *arXiv preprint arXiv:2507.12979*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Preprint*, arXiv:1706.03762.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model. *Preprint*, arXiv:2401.02385.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

## A Derivation: Break-even Condition for Test-Time Compute (TTC) with Continuous Refreshing

Define the following quantities per unit time (e.g., per year):

- $\mathrm{FLOPs}_{\text{train}}$: training FLOPs for the baseline (no TTC).
- $\mathrm{FLOPs}_{\text{infer}}$: inference FLOPs for the baseline (no TTC).
- $r \in (0, 1]$: ratio of training FLOPs under TTC to baseline training FLOPs, i.e. $r = \frac{\mathrm{FLOPs}_{\text{train}}^{\text{TTC}}}{\mathrm{FLOPs}_{\text{train}}}$.
- $\lambda_{\text{TTC}} > 0$: inference cost multiplier when using TTC (per-token inference cost relative to the "no TTC" baseline).
- $\lambda_{\text{base}} \geq 1$: inference cost multiplier already present in the baseline deployment (equals 1 when the baseline has no extra inference overhead).
- $f > 0$: refresh frequency (number of training+deployment cycles per unit time). For example, if the model is retrained and deployed three times per year, then f =3. For the algebra below $f$ cancels and therefore does not affect the bound.

Total FLOPs per unit time for the baseline and for TTC-aware operation are

$$C_{\text{baseline}} = f \cdot (\mathrm{FLOPs}_{\text{train}} + \lambda_{\text{base}}\, \mathrm{FLOPs}_{\text{infer}}), \tag{6}$$

$$C_{\text{TTC}} = f \cdot (r\, \mathrm{FLOPs}_{\text{train}} + \lambda_{\text{TTC}}\, \mathrm{FLOPs}_{\text{infer}}). \tag{7}$$

Here we assume that we can train and/or deploy the underlying model with TTC controlled by $\lambda_{\text{base}}$ or $\lambda_{\text{TTC}}$.
Requiring TTC to be no more expensive than the baseline gives

$$C_{\text{TTC}} \leq C_{\text{baseline}}.$$

Cancel $f$ (assuming $f > 0$):

$$r\, \mathrm{FLOPs}_{\text{train}} + \lambda_{\text{TTC}}\, \mathrm{FLOPs}_{\text{infer}} \leq \mathrm{FLOPs}_{\text{train}} + \lambda_{\text{base}} \mathrm{FLOPs}_{\text{infer}}$$

Thus the FLOPs-level break-even condition is

$$\boxed{(1 - r)\, \mathrm{FLOPs}_{\text{train}} \;\geq\; (\lambda_{\text{TTC}} - \lambda_{\text{base}})\, \mathrm{FLOPs}_{\text{infer}}.}$$

Next express the bound in terms of token counts. Let $N_{\text{train}}$ be the number of training tokens per refresh (baseline) and $N_{\text{infer}}$ the number of inference tokens served before the next refresh. Using the common approximation that one training token costs roughly $6\times$ the FLOPs of one inference token (accounting for forward + backward + optimizer) (Kaplan et al., 2020; Anil et al., 2023),

$$\frac{\mathrm{FLOPs}_{\text{train}}}{\mathrm{FLOPs}_{\text{infer}}} \approx \frac{6\, N_{\text{train}}}{N_{\text{infer}}}.$$

Substitute this ratio into the FLOPs inequality:

$$(1 - r) \cdot \frac{6\, N_{\text{train}}}{N_{\text{infer}}} \cdot \mathrm{FLOPs}_{\text{infer}} \geq (\lambda_{\text{TTC}} - \lambda_{\text{base}})\, \mathrm{FLOPs}_{\text{infer}}$$

Cancel $\mathrm{FLOPs}_{\text{infer}} > 0$ and solve for $N_{\text{infer}}$:

$$(1 - r) \cdot 6\, N_{\text{train}} \;\geq\; (\lambda_{\text{TTC}} - \lambda_{\text{base}})\, N_{\text{infer}}$$

$$N_{\text{infer}} \;\leq\; \frac{6\,(1 - r)}{\lambda_{\text{TTC}} - \lambda_{\text{base}}}\, N_{\text{train}}.$$

Setting $\lambda_{\text{base}} = 1$ recovers the special case where the baseline has no extra inference overhead:

$$\boxed{N_{\text{infer}} \leq \frac{6(1 - r)}{(\lambda - 1)} \cdot N_{\text{train}}}$$

**Remarks:**

- The denominator $\lambda_{\text{TTC}} - \lambda_{\text{base}}$ must be positive for the bound to be meaningful. If $\lambda_{\text{TTC}} \leq \lambda_{\text{base}}$, the accuracy of the intermediate checkpoint with TTC will not be better than the fully trained checkpoint.
- For $\lambda_{\text{base}} > 1$, the denominator $(\lambda_{\text{TTC}} - \lambda_{\text{base}})$ decreases, which increases the upper bound on $N_{\text{infer}}$. This means that as the baseline incurs more inference overhead, TTC-aware training allows proportionally more inference tokens before the break-even point is reached. For this reason, we set $\lambda_{\text{base}}$ to 1 to minimize the overhead of the baseline.
- This bound gives the maximum number of inference tokens that can be served between refreshes while keeping the total FLOPs (training plus deployment) at or below the non-TTC baseline.

## B Curve Fitting Results and Analysis

Figures 7 and 8 illustrate exponential curve fitting on the checkpoints, showing minimal fitting error. Because we exclude non-monotonic points in curve fitting, we remove the expected fluctuations in the validation accuracy. Shown in Table 5, we observe that removing these fluctuations result in better FLOPs savings – from 56.8% to 90.7% – in TinyLlama on HumanEval. Thus, our curve fitting procedure for training is more stable and has more FLOPs savings while at least maintaining the accuracy. Additionally, Figure 9 presents sigmoid curve projections, which closely track TTC accuracy across checkpoints at different FLOP levels.

To further guard against potential fitting failures, we incorporate *patience*, a standard hyperparameter in early stopping. Figure 6 illustrates the impact of varying patience on the solve rate improvement for TTC Pass@16—relative to the fully trained checkpoint—and the corresponding training FLOPs savings for TinyLlama on the HumanEval dataset. A clear tradeoff emerges: increasing patience generally improves the solve rate but also decreases the fraction of training FLOPs saved. While larger patience values yield slightly higher improvements, the FLOPs savings decrease, resulting in diminishing returns in computational efficiency. An intermediate patience value of around 10 appears optimal, providing substantial solve rate gains while keeping training FLOPs within a reasonable range. This underscores the importance of carefully tuning patience to balance performance and efficiency.
To further verify robustness, we also repeated the same experiments at temperatures 0.7, 0.8, and 0.9, each with three seeds. The resulting average standard deviation across checkpoints remains close to 0.1, confirming stability across decoding settings as well.

## C TTC Latency Overhead

A common concern is that test-time compute (TTC) increases deployment latency because decoding is repeated $K$ times (e.g., Pass@$K$, majority vote, or verifier-based search). In our framework, this overhead is explicitly controlled in two ways. First, the early-stopping procedure estimates the *smallest* TTC budget $K^*$ that meets the target accuracy while still satisfying the total-compute constraint in Eq. 3. Empirically, $K^*$ is modest: on both TinyLlama/HumanEval and Qwen3-30B/GSM8k, intermediate checkpoints can already match or exceed the fully trained baseline with small $K$ (Figure 3), and on FineMath/Math-500, a **10B-token** checkpoint with verifier-based DVTS at $K{=}4$ reaches **24.73%**, outperforming a **50B-token** checkpoint with $K{=}2$ (**18.28%**) (Table 4). These results indicate that very large $K$ is not necessary for deployment; TTC gains are realizable at low-to-moderate $K$.

Second, we quantify the *end-to-end* overhead through the break-even bound in Eq. 5, which guarantees when "TTC-aware training + deployment" remains cheaper than training and deploying the fully trained baseline. This bound uses $\lambda$ (the inference-cost multiplier induced by TTC) and $r$ (the training-FLOPs ratio after early stopping). Table 1 shows that even with non-trivial TTC overhead, the training savings can fund a large amount of downstream inference before the next refresh: for example, with a modest overhead $\lambda{\approx}1.2\times$, the bound permits **54T–72T** inference tokens, corresponding to **52.7B–70.3B** samples of length 1024 tokens. Even for larger overheads (e.g., $\lambda{=}8$ or $\lambda{=}16$), the break-even point remains on the order of **0.96T–1.54T** inference tokens (Table 1).

Finally, the practical overhead during training is negligible: curve fitting (exponential/sigmoid over a few points) is closed-form and run only when patience triggers a check, while TTC sampling is *fully accounted for* by the break-even analysis. Overall, TTC-aware early stopping trades a small, controlled inference-time multiplier for a large reduction in training compute, enabling faster model development and more frequent refresh cycles. Economically, even in our **1.1B** TinyLlama experiment, TTC-aware early stopping reduces the effective training from **1,657 GPU-days** to at least **127 GPU-days** (Pass@8), roughly cutting cost from **135k** to **10k** on Azure V100 pricing; since training cost scales approximately linearly with model size, the relative savings become even more meaningful at **70B–400B** scale.

## D Robustness Analysis of the TTC $K^*$ Estimator

The proposed TTC-aware early stopping method uses a lightweight estimator to efficiently select the TTC budget $K^*$. Specifically, rather than exhaustively evaluating a large set of $K$ values, we fit a three-parameter sigmoid using a small set of observed TTC accuracies at $K \in \{1, 2, 4\}$ and use the fitted curve to estimate the smallest $K^*$ that

satisfies accuracy and compute constraints. Since this procedure involves extrapolation from small-$K$ observations, we further evaluate its robustness and prediction error.

We evaluated the extrapolation accuracy of the sigmoid-based TTC estimator on TinyLlama-1B using the HumanEval benchmark. For each checkpoint, we fit the sigmoid using TTC accuracies measured at $K \in \{1, 2, 4\}$ and then predict the accuracy at the target TTC setting. Table 7 reports the actual TTC accuracy, predicted TTC accuracy, and absolute prediction error across 14 checkpoints (see Figure 9).

Table 7: Extrapolation error of the sigmoid-based TTC $K^*$ estimator on TinyLlama-1B with HumanEval. The estimator fits $K \in \{1, 2, 4\}$, and is evaluated against actual measured TTC accuracy. $K^* = 8$ in this case.

| Train FLOPs | Actual (%) | Predicted (%) | Abs. Error (%) |
|---|---|---|---|
| $7.31\times10^{19}$ | 1.829 | 2.318 | 0.489 |
| $7.60\times10^{20}$ | 6.095 | 4.739 | 1.356 |
| $9.94\times10^{20}$ | 8.537 | 6.122 | 2.414 |
| $1.92\times10^{21}$ | 9.146 | 7.665 | 1.482 |
| $2.84\times10^{21}$ | 6.707 | 4.908 | 1.799 |
| $3.68\times10^{21}$ | 12.195 | 9.185 | 3.011 |
| $4.60\times10^{21}$ | 9.756 | 7.862 | 1.894 |
| $7.31\times10^{21}$ | 9.146 | 6.910 | 2.237 |
| $8.27\times10^{21}$ | 9.756 | 7.786 | 1.970 |
| $9.20\times10^{21}$ | 10.366 | 7.665 | 2.701 |
| $1.01\times10^{22}$ | 9.756 | 7.835 | 1.921 |
| $1.10\times10^{22}$ | 8.533 | 6.397 | 2.136 |
| $1.46\times10^{22}$ | 10.363 | 8.115 | 2.248 |
| $1.83\times10^{22}$ | 10.362 | 8.589 | 1.773 |

Across these 14 checkpoints, the estimator achieves a mean absolute error (MAE) of $1.96\%$, a root mean squared error (RMSE) of $2.05\%$, and a Pearson correlation of $r = 0.978$ between predicted and actual TTC accuracies. These results show that the sigmoid fit captures the overall TTC trend across checkpoints despite relying only on small-$K$ observations.

We also observe a small underestimation bias of approximately $+1.89\%$, meaning that the predicted TTC accuracy is typically lower than the actual measured value. This behavior makes the $K^*$ selection conservative: the algorithm is less likely to stop early based on an overly optimistic extrapolation. Importantly, the fitted curve is used only to select a candidate $K^*$ during early stopping. The final reported results are based on actual measured TTC accuracy rather than extrapolated values. Therefore, the estimator affects the efficiency of the selection procedure, but not the validity of the final reported performance.

## E Comparison with Early Stopping Baselines with Validation and Discussion on TTC Evaluation Overhead

A practical concern in TTC-aware early stopping is that checkpoint selection itself may introduce additional compute, since validation and TTC evaluation are performed periodically during training. To make this cost explicit, we compare TTC-aware early stopping against several early stopping baselines on TinyLlama-1B with HumanEval, while including the validation and TTC evaluation overhead in the total compute.

As shown in Table 8, we report total cost in V100-days, including periodic evaluation overhead for validation-based early stopping methods. For clarity, we explicitly define the stopping criteria used by each baseline:

- **No early stopping (ES):** Training proceeds for the full predefined budget.
- **Early stopping based on validation performance (uses Pass@1):** Stopping is triggered based on validation performance computed using standard single-sample inference (Pass@1) according to a fixed patience criterion.
- **Early stopping based on training loss:** Training is stopped when the training loss ceases to improve according to a fixed patience criterion. This method incurs no additional evaluation cost. We obtain the training losses from the published TinyLlama logs[1].
- **Early stopping based on validation performance (with TTC):** Stopping is based on validation performance measured using Pass@8 with a fixed patience criterion, which captures test-time computation (TTC) effects.

Validation-based methods incur additional compute due to periodic evaluation. In our setup, Pass@1 validation costs approximately $0.03037$ V100-days per evaluation, while Pass@8 TTC validation costs approximately $0.24296$ V100-days per evaluation. To be conservative, we assume a worst-case evaluation setting in which Pass@8 is

[1] https://wandb.ai/lance777/lightning_logs/reports/metric-train_loss-23-09-04-23-38-15---Vmldzo1MzA4MzIw?accessToken=5eu2sndit2mo6eqls8h38sklcgfwt660ek1f2czlgtqjv2c6tida47qm1oty8ik9

implemented as eight independent Pass@1 generations. Accordingly, we approximate the evaluation cost as $8\times$ the Pass@1 V100-day cost, ignoring potential efficiency gains from prompt reuse (e.g., KV caching) or batching. Accounting for such optimizations is left for future work.

Table 8: Early stopping (ES) baselines on TinyLlama-1B with HumanEval, including validation and TTC evaluation overhead. Total V100-days include periodic evaluation cost, except for train-loss-based early stopping, which does not require validation evaluation.

| **Strategy** | **V100-days** | **Savings** | **Pass@1** | **Pass@8** |
|---|---|---|---|---|
| No ES | 1630.8 | – | 9.15% | 12.70% |
| Val ES ($p$=50) | 1657.4 | −1.6% | 9.15% | 12.10% |
| Val ES ($p$=10) | 113.5 | +93.0% | 6.09% | 7.73% |
| Train ES ($p$=10) | 259.0 | +84.1% | 6.09% | 7.50% |
| Train ES ($p$=50) | 579.9 | +64.4% | 6.09% | 9.50% |
| TTC ES ($K^*$=8) | 155.9 | +90.4% | 5.50% | 9.70% |

Table 8 shows that TTC-aware early stopping achieves the most favorable compute–accuracy trade-off among the strategies considered, even under the conservative assumption that Pass@8 TTC evaluation is performed at every validation point. Compared with no early stopping, it reduces total cost from 1630.8 to 155.9 V100-days (a 90.4% reduction) while retaining a Pass@8 of 9.70%.

The other strategies each illustrate a different failure mode. Validation-based early stopping with patience $= 50$ is too conservative: the added evaluation overhead actually *increases* total cost by 1.6% relative to no early stopping, with no accuracy gain. Tightening to patience $= 10$ recovers aggressive savings (93.0%) but collapses Pass@8 to 7.73%, since the standard validation loss is a poor proxy for downstream TTC performance. Training-loss early stopping sidesteps validation overhead entirely, but does not directly optimize the deployment-relevant TTC metric: matching TTC-aware ES's Pass@8 (9.50% vs. 9.70%) requires patience $=$ 50 and 579.9 V100-days (roughly $3.7\times$ the compute of TTC-aware ES at comparable accuracy).

TTC-aware ES explicitly uses TTC validation performance for checkpoint selection, which is why it dominates on Pass@8 at low compute. Overall, this analysis confirms that the additional validation and TTC evaluation overhead does not offset the main training-compute savings. Even under conservative TTC evaluation accounting, TTC-aware early stopping remains substantially cheaper than full training while preserving competitive TTC performance.

# F Details of TTC-Aware Exponential-Fit Early Stopping

Algorithm 1 implements a deployment-aligned early-stopping rule that decides when to stop training and how much test-time compute to use at the selected checkpoint. Instead of optimizing for the final checkpoint at the full training budget $B$, the algorithm searches for an intermediate checkpoint at FLOPs $t$ paired with a minimal TTC budget $K^*$ that (i) meets or exceeds the predicted fully-trained accuracy and (ii) is cheaper in total compute than training to $B$ and deploying with standard decoding.

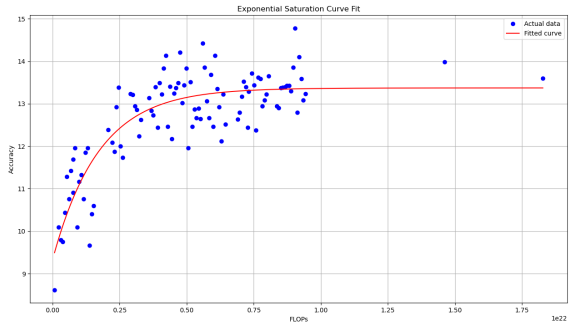
**Notation and objective.** At each training time $t$, we consider two costs: training FLOPs $F_{\text{tr}}(t)$ and inference FLOPs under TTC $F_{\text{inf}}(t, K)$, where $K$ denotes the number of samples/decoding attempts used by the TTC method (e.g., Pass@$K$, majority vote, DVTS-based search). We also denote the TTC-validated accuracy at time $t$ using budget $K$ by $\text{Acc}_t^{\text{ttc}}(K)$, which may be estimated on a held-out validation set. The core goal is to find a *compute-dominating* TTC checkpoint, i.e.,

$$\begin{gathered} F_{\text{tr}}(t) + F_{\text{inf}}(t, K) < F_{\text{tr}}(B) + F_{\text{inf}}(B, 1), \\ \textit{and} \\ \text{Acc}_t^{\text{ttc}}(K) \geq Acc(B). \end{gathered} \tag{8}$$

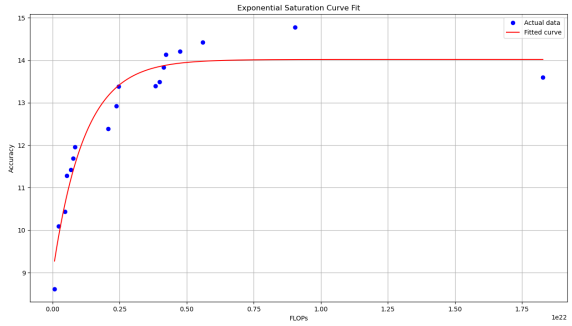
where $Acc(B)$ is the accuracy of the (unknown) fully trained checkpoint. Since $Acc(B)$ is not available before completing training, we estimate it via learning-curve fitting.

**Step 1: Exponential learning-curve fit to forecast $f(B)$.** Line 2 defines an exponential saturation model $f(x) = a(1 - e^{-bx}) + c$, where $x$ is the cumulative training FLOPs (or steps mapped to FLOPs). At each iteration $t$ (lines 4–6), the algorithm fits $f(\cdot)$ to the observed validation accuracies $\{val\_acc\}$ collected so far and uses the fitted curve to forecast the baseline final accuracy at the full budget, $f(B)$. This forecast acts as a surrogate for the fully trained checkpoint performance and is updated online as new checkpoints are observed.

**Step 2: Selecting the minimal TTC budget $K^*$.** Given the current checkpoint at $t$, the algorithm chooses the *smallest* TTC budget $K^*$ that satisfies

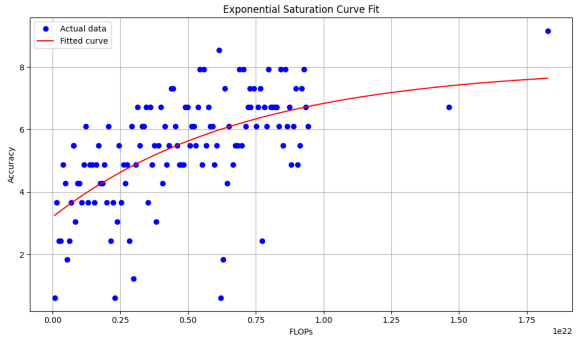



(a) Curve fit across all checkpoints with fluctuations.

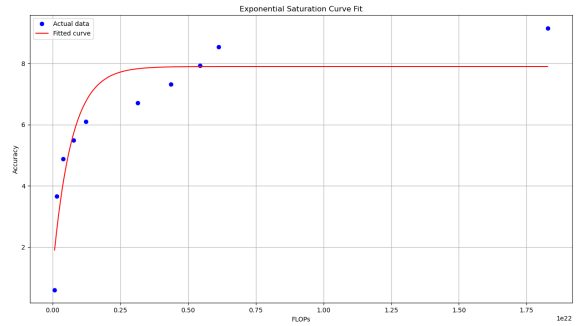



(b) Curve fit across all checkpoints without fluctuations.

Figure 7: Exponential saturation curve fitting results for **TinyLlama on DROP**. (a) shows the fit when using all available checkpoints. (b) shows the fit when using all available checkpoints and ignoring fluctuations. In all cases, the red curve represents the fitted function and the blue points denote actual data, where each point corresponds to a TinyLlama checkpoint evaluated on the DROP dataset. After fluctuation filtering, the relative error between the fitted checkpoint and the final observed checkpoint is as low as +0.4%




(a) Curve fit across all checkpoints with fluctuations.




(b) Curve fit across all checkpoints without fluctuations.

Figure 8: Exponential saturation curve fitting results. (a) shows the fit when using all available checkpoints for **TinyLlama on HumanEval**. (b) shows the fit when using all available checkpoints and ignoring fluctuations. In all cases, the red curve represents the fitted function and the blue points denote actual data, where each point corresponds to a TinyLlama checkpoint evaluated on the HumanEval dataset. After fluctuations filtering, relative error between the fitted checkpoint and the last actual checkpoint = -1.8%

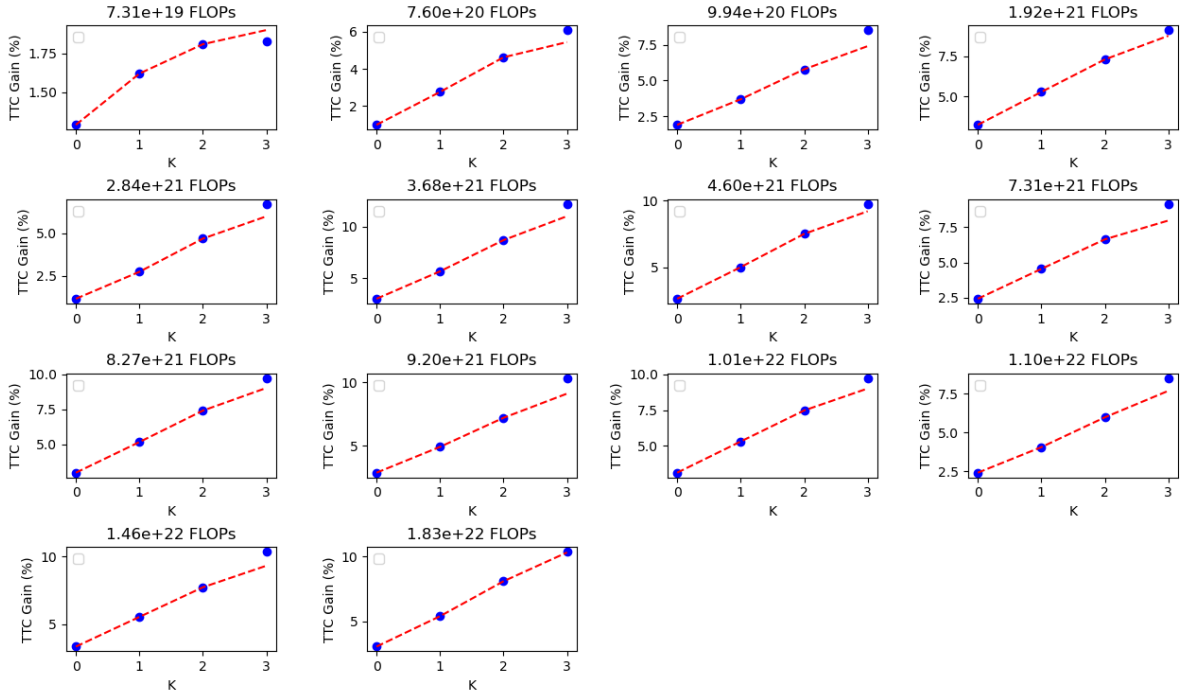



Figure 9: Sigmoid fitting for the parameter $K$ in TTC at different FLOPs. TTC gain is recorded for $K = 1, 2, 4$ and the fitted curve is used to estimate accuracy at $K = 8$ on TinyLlama-1B with the HumanEval dataset. Note that X-axis indicates the index of the K rather than the actual value.

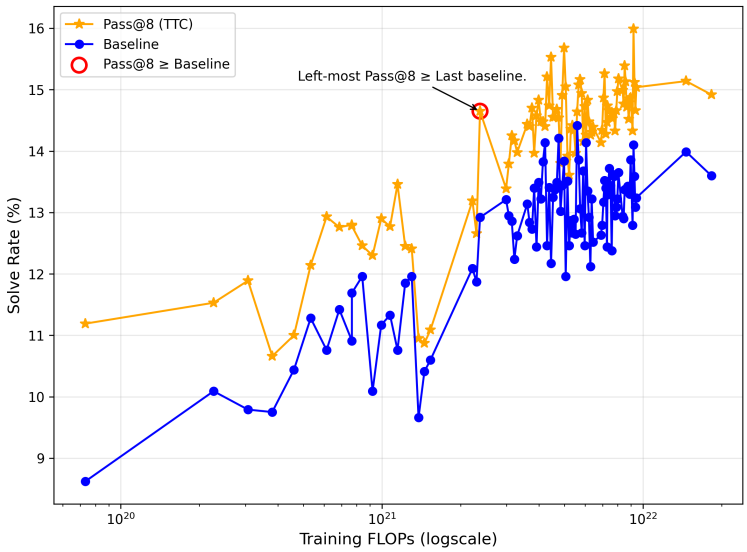



Figure 10: Results of TTC-aware training on the DROP dataset using TinyLlama-1B. Similar to Figure 1, applying TTC allows early stopping while saving up to 92% of the training FLOPs without sacrificing final accuracy. This demonstrates that TTC awareness generalizes across datasets and is not limited to a single benchmark.

two constraints (line 8):

$$F_{\text{tr}}(t) + F_{\text{inf}}(t, K^*) < F_{\text{tr}}(B) + F_{\text{inf}}(B, 1), \tag{9}$$

$$\text{Acc}_t^{\text{ttc}}(K^*) \geq f(B). \tag{10}$$

The compute constraint in Eq. (9) enforces that "stop at $t$ + deploy with TTC" is strictly cheaper than "train to $B$ + deploy with standard decoding." The accuracy constraint in Eq. (10) enforces that TTC at checkpoint $t$ meets (or exceeds) the predicted fully trained accuracy. By minimizing $K^*$, the algorithm avoids unnecessarily large TTC overhead and directly targets the smallest deployment-time cost that achieves the desired quality.

**Step 3: Tracking the best TTC checkpoint.** Once $K^*$ is determined, the algorithm evaluates (or looks up) TTC validation accuracy at that $K^*$, denoted $val\_ttc\_acc_t[K^*]$. If this value improves upon the best TTC validation accuracy seen so far, $best\_ttc\_val\_acc$, we reset the patience counter (line 10) and update $best\_ttc\_val\_acc$ (line 12). This ensures we retain the best *deployable* checkpoint under the TTC budget dictated by the inequality and target accuracy.

**Step 4: Patience-based stopping criterion.** The stopping condition (lines 14–17) triggers when the algorithm has observed no TTC-improving checkpoint for $p$ consecutive checks *after* the TTC performance has reached the forecasted baseline. Concretely, if $best\_ttc\_val\_acc \geq f(B)$, the patience counter increments; otherwise it is reset (line 14). When $patience\_counter \geq p$, training stops and the algorithm returns the best TTC checkpoint found so far (line 16). This design prevents premature stopping before TTC catches up to the predicted final baseline, while still terminating soon after the training trajectory saturates in a deployment-relevant sense.

**Practical notes.** (i) The fit is performed on a small set of observed points and updated online; in practice, it is invoked only when a patience check is performed. (ii) $F_{\text{inf}}(t, K)$ can be measured directly (e.g., wall-clock or FLOPs) or modeled as approximately linear in $K$ for sampling-based TTC, which is sufficient for selecting $K^*$. (iii) The algorithm is agnostic to the particular TTC procedure: $\text{Acc}_t^{\text{ttc}}(K)$ may come from Pass@$K$, majority vote, verifier-guided search, or other TTC policies, as long as accuracy increases monotonically (or approximately monotonically) with $K$.

## G Inference FLOPs Accounting for TTC Methods

We explicitly account for the inference cost introduced by different TTC methods when comparing TTC-aware early stopping with full training. Let $F_{\text{gen}}$ denote the forward-only generation FLOPs for one sampled output from the base model, and let $F_{\text{verifier}}$ denote the forward FLOPs required by a

verifier or reward model for scoring one candidate output. For LLaMA-style models with grouped-query attention (GQA), $F_{\text{gen}}$ is computed using standard Transformer forward FLOPs, including attention projections, GQA attention, MLP layers, normalization, and output projection. The TTC inference cost is then derived from the number of generation and verifier forward passes required by each method.

For Pass@$K$, the model generates $K$ independent samples. Therefore, the inference cost scales approximately linearly with $K$:

$$F_{\text{inf}}^{\text{Pass@}K} \approx K \cdot F_{\text{gen}}. \tag{11}$$

Majority voting also requires $K$ sampled generations, followed by a lightweight aggregation step. Since the aggregation cost is negligible compared with model forward passes, we use the same leading-order cost:

$$F_{\text{inf}}^{\text{Majority}} \approx K \cdot F_{\text{gen}}. \tag{12}$$

For verifier-based TTC, the system first generates $K$ candidate outputs and then scores these candidates using a verifier or reward model. The total cost includes both candidate generation and verifier evaluation:

$$F_{\text{inf}}^{\text{Verifier}} \approx K \cdot F_{\text{gen}} + K \cdot F_{\text{verifier}}. \tag{13}$$

For Diverse Tree Search (DVTS), the cost depends on the number of tree node expansions and verifier calls. Let $N_{\text{expand}}$ denote the total number of model-generation expansions and $N_{\text{verify}}$ denote the number of verifier evaluations. The corresponding inference cost is:

$$F_{\text{inf}}^{\text{DVTS}} \approx N_{\text{expand}} \cdot F_{\text{gen}} + N_{\text{verify}} \cdot F_{\text{verifier}}. \tag{14}$$

For compute-optimal search, which selects the best-performing method among the evaluated TTC policies, we account for the cost of the underlying selected policy:

$$F_{\text{inf}}^{\text{CTS}} = F_{\text{inf}}^{m^*}, \qquad m^* = \arg\max_{m \in \mathcal{M}} \text{Acc}^m, \tag{15}$$

where $\mathcal{M}$ is the set of candidate TTC methods and $m^*$ is the selected method under the given compute budget.

This accounting is used consistently in the TTC-aware early stopping inequality and in the break-even analysis. In particular, the total cost of a TTC-aware checkpoint at training step $t$ is measured as:

$$F_{\text{total}}(t, K) = F_{\text{tr}}[t] + F_{\text{inf}}[t, K], \tag{16}$$

and it is compared against the fully trained baseline:

$$F_{\text{tr}}[t] + F_{\text{inf}}[t, K^*] < F_{\text{tr}}[B] + F_{\text{inf}}[B, 1]. \tag{17}$$

The use of $K = 1$ for the fully trained baseline is intentionally conservative because it gives the baseline the lowest inference cost. If a deployment also uses TTC for the fully trained checkpoint, the comparison becomes:

$$F_{\text{tr}}[t] + F_{\text{inf}}[t, K^*] < F_{\text{tr}}[B] + F_{\text{inf}}[B, K_{\text{deploy}}], \tag{18}$$

where $K_{\text{deploy}}$ is the deployment TTC policy used by the fully trained baseline. For sampling-based TTC, $F_{\text{inf}}[B, K_{\text{deploy}}] \geq F_{\text{inf}}[B, 1]$, so the $K = 1$ baseline used in the main comparison is the stricter condition.

Table 9: Pretraining configuration for TinyLLaMA Training in LLaMA-Factory (Zheng et al., 2024).

| Category | Configuration |
|---|---|
| Model | quantization_method: None<br>train_from_scratch: True<br>disable_gradient_checkpointing: False |
| Method | stage: pt<br>do_train: True<br>finetuning_type: full<br>deepspeed: examples/deepspeed/ds_z3_config.json |
| Dataset | dataset: cosmopedia-v2, From HuggingFaceTB/smollm-corpus<br>cutoff_len: 2048<br>preprocessing_num_workers: 120<br>streaming: False, packing: True<br>overwrite_cache: True<br>dataloader_num_workers: 8 |
| Training | per_device_train_batch_size: 16<br>gradient_accumulation_steps: 4<br>learning_rate: 2.0e-5<br>lr_scheduler_type: cosine<br>warmup_ratio: 0.005<br>fp16: True, bf16: False<br>ddp_timeout: 180000000<br>num_train_epochs: 1 |
| Evaluation | val_size: 5000<br>per_device_eval_batch_size: 1<br>eval_strategy: steps<br>eval_steps: 200 |
| Output | logging_steps: 1, save_steps: 200<br>plot_loss: True<br>overwrite_output_dir: True<br>save_total_limit: 500<br>report_to: tensorboard |