LEARNING TO RETOKENIZE: IMPROVING DOWNSTREAM PERFORMANCE OF FROZEN LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Language modelling in LLMs begins with tokenization: splitting the input string into tokens drawn from a fixed vocabulary according to predefined rules. Because the tokenizer is trained on a generic corpus before model pre-training and is not tailored to any downstream task, its tokenization strategy can be ill-suited to certain domains or tasks, hindering performance. Adapting tokenization per task could help, but retraining the tokenizer on a new corpus likely changes the vocabulary and requires model adaptation, which is costly in general and infeasible for closed-weight models. A tokenizer that reuses the original vocabulary, however, could be applied to the model seamlessly. In this work, we explore training a task-specific tokenizer that is constrained to the original vocabulary of a given pre-trained LLM. To this end, we propose the retokenizer, a sequence-to-sequence model that maps an input token sequence to a new one within the original vocabulary, while preserving the underlying string content. During training, our retokenizer learns from a task-specific reward and only requires black-box access to the language model, making the approach suitable for closed-weight models. Across arithmetic, character manipulation, and code understanding tasks, our learned retokenization improves accuracy over the default tokenization by 10% on average. Moreover, we show that in some settings the retokenizer uncovers intuitive task-dependent tokenization rules, such as right-to-left digit tokenization for arithmetic. These learned rules suggest that, given the right objective, effective tokenization strategies can be discovered automatically rather than designed by hand.

1 INTRODUCTION

Despite their strong performance across many domains, large language models (LLMs) can struggle with seemingly simple tasks that require fine-grained access to character-level, numerical, or syntactic structure. Tokenization is a common culprit: LLMs process inputs as sequences of subword tokens, whose boundaries may obscure the structure relevant to a given task (Chai et al., 2024; Wallace et al., 2019). The famous strawberry problem illustrates this mismatch (Cosma et al., 2025): many pretrained models incorrectly answer "2" when asked to count the occurrences of `r` in "strawberry". Although the task requires identifying individual characters, the tokenizer may group them into a single unit; for example, the Llama 3 tokenizer (Grattafiori et al., 2024) represents `␣strawberry` as one token.

This mismatch is perhaps unsurprising: the tokenizer is designed independently of the downstream task on which the language model will be used. Subword tokenizers, such as Byte-Pair Encoding (BPE) (Gage, 1994; Sennrich et al., 2016), are generally built from the statistics of generic corpora before language-model pretraining. The resulting tokenization strategy is not tailored to any particular downstream task or input distribution, and may be poorly suited to settings requiring access to structures not important when the tokenizer was constructed, especially on out-of-domain inputs.

Existing approaches address these limitations generally by changing how the language model is built or trained. For example, by replacing subword tokenization with character- or byte-level models (Clark et al., 2022; Xue et al., 2022), modifying the architecture to better recover fine-grained information from existing subword tokens (Cosma et al., 2025), or learning a task-specific tokenizer

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.

that introduces new tokens to the vocabulary set and fine-tuning the model to support them (Liu et al., 2023). These approaches require modifying the language model's weights, which can be costly or infeasible for existing pretrained models, particularly closed-weight models. This motivates a different question: *if both the language model and its vocabulary are kept fixed, can performance be improved by changing only how each input is tokenized?*

Even with the model and vocabulary fixed, the same input string can admit multiple valid tokenizations, and the choice of segmentation can affect downstream performance. While a pretrained BPE tokenizer selects one canonical segmentation, prior work has shown that alternatives can improve performance: character-level tokenization benefits character manipulation and code-understanding tasks, while *right-to-left* digit tokenization benefits arithmetic (Zheng et al., 2025a; Singh & Strouse, 2024). However, manually designing such rules requires task-specific knowledge, can substantially increase sequence length, and may not transfer across language models. We therefore ask whether useful tokenization strategies can instead be learned directly from downstream performance.

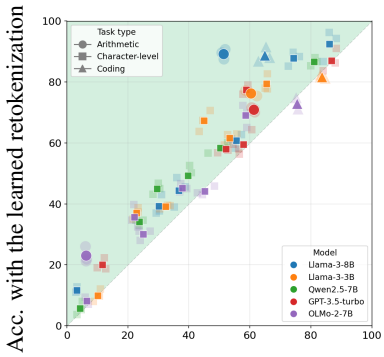

Figure 1: **Learned retokenization improves downstream accuracy while keeping the LLM and its vocabulary fixed.** Accuracy with the default canonical BPE tokenizer versus the learned retokenizer, trained based on the frozen LLM's downstream performance. Each dark point represents one task–model setting reported in Table 1, with the lighter points showing the individual seeds. See Sections 1 and 3 for details.

To this end, we introduce a *retokenizer*, a trainable sequence-to-sequence model that takes the canonical token sequence as input and generates an alternative token sequence to pass to the frozen language model. Its output uses the language model's existing vocabulary and is constrained to decode to the original input string, allowing the retokenizer to change the segmentation while preserving the input's content. We train the retokenizer with reinforcement learning, using the quality of the frozen model's response to a sampled tokenization as the reward. Training requires only the model's outputs, without access to its parameters, gradients, or hidden states, making the approach applicable to both open- and closed-weight models.

We evaluate learned retokenization across arithmetic, character-manipulation, and code-understanding tasks using both open- and closed-weight language models. As summarized in Figure 1 and detailed in Table 1, learned retokenization improves over canonical tokenization in most evaluated settings, including those using the closed-weight GPT-3.5-turbo-instruct model. Compared with task-specific tokenization rules from prior work, the learned strategy adapts to both the task and the language model: in some cases it recovers intuitive rules such as right-to-left digit tokenization, while in others it discovers less obvious tokenization strategies that nevertheless improve performance. Our contributions are:

1. We introduce a trainable retokenizer that improves downstream performance of a frozen language model without changing its vocabulary or parameters.

2. We formulate retokenization as a reinforcement-learning problem that requires only black-box access to the language model, enabling application to closed-weight models.

3. We evaluate learned retokenization across arithmetic, character-manipulation, and code-understanding tasks, showing improvements over canonical tokenization and competitive performance with hand-designed tokenization alternatives.

4. We show that the effective tokenization strategies can be model-dependent: the retokenizer sometimes recovers intuitive rules, while in other cases it discovers less obvious strategies that improve performance.

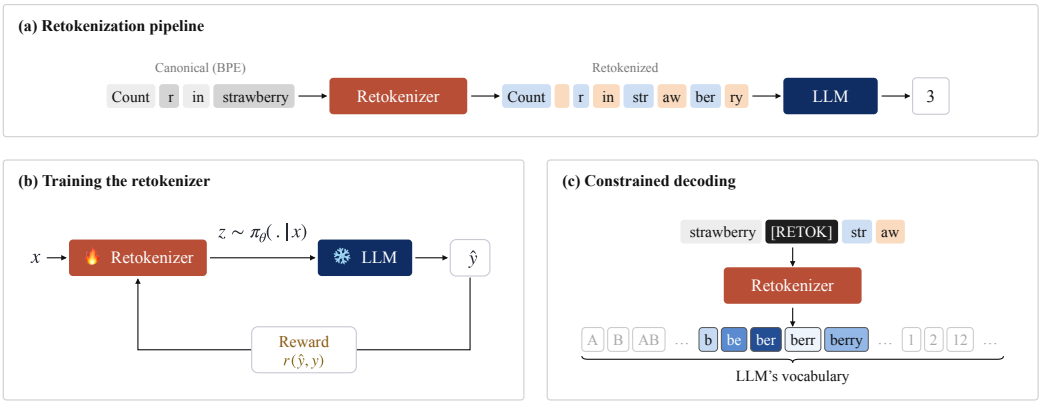

Figure 2: **Overview of learned retokenization.** (a) The retokenizer replaces the model's canonical tokenization with an alternative tokenization, with the goal of finding a segmentation better suited to the downstream task while keeping the LLM and its vocabulary unchanged. (b) The retokenizer is trained to maximize the task reward: at training time, it samples a retokenization, the frozen LLM generates a response, and a reward computed from the response is used to update the retokenizer. (c) Constrained decoding ensures that the complete token sequence decodes exactly to the original input string by masking the tokens that are not consistent with the input string. See Sec. 2 for details.

2 Problem Formulation

Given a pretrained language model and a downstream task, our goal is to improve task performance by changing only the tokenization strategy, while keeping the model parameters and vocabulary fixed. Our premise is that the model's default tokenization may not be the most effective representation for the task. We therefore seek to learn which tokenizations work best, using task performance as the learning signal.

2.1 Setup

Let $\mathcal{V}$ be the fixed vocabulary of a pretrained language model f. A tokenization of an input string $\boldsymbol{x}$ is valid if concatenating its tokens recovers $\boldsymbol{x}$ exactly. We write the set of valid tokenizations as

$$\mathcal{T}_{\mathcal{V}}(\boldsymbol{x}) = \{\boldsymbol{z} = (z_1, \ldots, z_m) \in \mathcal{V}^* : z_1 \circ \cdots \circ z_m = \boldsymbol{x}\},$$

where $\mathcal{V}^*$ is the set of all finite token sequences over $\mathcal{V}$ and $\circ$ denotes string concatenation.

The model's canonical tokenizer chooses one tokenization $\boldsymbol{z}_{\text{can}}(\boldsymbol{x}) \in \mathcal{T}_{\mathcal{V}}(\boldsymbol{x})$. Other tokenizations in this set preserve the input string but may change the model's performance on the task. We introduce a trainable *retokenizer* that learns to select tokenizations from $\mathcal{T}_{\mathcal{V}}(\boldsymbol{x})$ using task performance as its learning signal. Our goal is for this strategy to improve performance on unseen inputs from the same task, while keeping the model parameters and vocabulary fixed.

2.2 Retokenizer Module

We parameterize the tokenization strategy with a trainable *retokenizer*, which selects how to segment each input for the pretrained language model. The retokenizer π_θ takes the canonical tokenization $\boldsymbol{z}_{\text{can}}(\boldsymbol{x})$ of an input string $\boldsymbol{x}$ and defines a distribution over valid tokenizations of that string:

$$\pi_\theta(\boldsymbol{z} \mid \boldsymbol{x}) = 0 \qquad \text{for all } \boldsymbol{z} \notin \mathcal{T}_{\mathcal{V}}(\boldsymbol{x}). \tag{1}$$

We implement π_θ as a decoder-only transformer (Radford et al., 2018) that generates tokens autoregressively:

$$\pi_\theta(\boldsymbol{z} \mid \boldsymbol{x}) = \prod_{t=1}^{m} \pi_\theta(z_t \mid \boldsymbol{z}_{\text{can}}(\boldsymbol{x}), \boldsymbol{z}_{<t}).$$

For simplicity, we write $\pi_\theta(\cdot \mid \boldsymbol{x})$, although the retokenizer is conditioned on $\boldsymbol{z}_{\text{can}}(\boldsymbol{x})$.

We enforce validity through constrained decoding. The retokenizer uses the same output vocabulary as the pretrained language model. At each decoding step, we mask all tokens except those that match a prefix of the remaining input string. Generation ends when the tokens cover the full string.

For example, when retokenizing `strawberry` (Figure 2-(c)), generating `str` and `aw` leaves the substring `berry`. The next token must be one of `b`, `be`, `ber`, `berr`, or `berry`, provided it belongs to $\mathcal{V}$. This procedure ensures that every generated sequence belongs to $\mathcal{T}_{\mathcal{V}}(\boldsymbol{x})$.

2.3 Training the Retokenizer

Given a training set $\mathcal{S} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{n}$ and a task-specific reward r, we optimize the retokenizer parameters θ to maximize the frozen language model's expected reward:

$$\max_{\theta} \mathcal{R}(\theta) := \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{z}_i \sim \pi_\theta(\cdot|\boldsymbol{x}_i)} \left[r\big(f(\boldsymbol{z}_i), \boldsymbol{y}_i\big) \right]. \tag{2}$$

We use policy-gradient methods (Williams, 1992; Shao et al., 2024; Zheng et al., 2025b) to adjust the probability of sampled tokenizations using their rewards. Each reward is treated as a scalar signal: we differentiate only the retokenizer's log-probabilities, not the language model.

Our rewards require only generated responses, enabling training with closed-weight models whose tokenizer and vocabulary are accessible and whose interface accepts token sequences directly.

3 Experiments

We evaluate whether task-driven retokenization improves downstream performance while keeping the language model fixed. We study five instruction-tuned models, including the closed-weight GPT-3.5-turbo-instruct, on tasks where prior work has shown tokenization to affect performance. Comparisons with canonical tokenization and relevant hand-designed baselines test whether effective tokenization strategies can be learned directly from task feedback.

3.1 Setup

Tasks. We consider three task families that require reasoning over fine-grained input components that canonical tokenization may group together or split apart: multi-digit addition, where the operands do not fit in a single token and the model should uncover individual digits and their place values; character-level reasoning, which involves accessing and composing individual characters that form multi-character tokens; and code understanding, which requires interpreting program elements such as variables, operators, and syntax. They therefore provide useful settings for evaluating whether retokenization alone can improve a frozen model without changing its vocabulary.

Multi-digit arithmetic. We generate expressions $a \pm b$, where both operands are seven-digit integers and the target is $\boldsymbol{y} = a \pm b$, using 5,000 expressions for training and 1,000 held-out expressions for evaluation. This setting builds on evidence that number tokenization can limit arithmetic performance and that retokenization alone can substantially improve it (Singh & Strouse, 2024; Zheng et al., 2025a).

Acronym generation. Following Zheng et al. (2025a), we ask the model to generate a phrase whose initial letters spell a given random acronym of five to seven letters. A response is correct if its word initials reproduce the acronym. We sample 2,000 acronyms for training and 500 for evaluation.

Character counting. Following Zheng et al. (2025a), we ask the model to count occurrences of a specified character in a word drawn from vocabulary tokens of 5–20 characters. We sample an absent character with probability 0.3 and a present character otherwise, using 1,000 word–character pairs each for training and evaluation.

CUTE benchmark. We use CUTE's character deletion, substitution, insertion, and contains-character subtasks (Edman et al., 2024).[1] Each subtask contains 1,000 prompts. We remove the

[1] We omit character swapping because near-zero canonical accuracy for some models provides too few positive-reward samples for reliable retokenizer training; see Section 6.

Table 1: **Downstream accuracy under canonical, hand-designed, and learned tokenization.** Accuracy (%) of frozen instruction-tuned models, averaged over three seeds; subscripts report standard deviations. Char. denotes character-level tokenization and R2L denotes right-to-left tokenization. Bold marks the highest mean for each model–task pair. Dashes indicate excluded settings (Section 3.1).

Task	Method	Llama-3.1 8B	Llama-3.2 3B	Qwen2.5 7B	OLMo-2 7B	GPT-3.5 turbo-instruct
Arithmetic	Canonical	$51.4_{0.6}$	$60.3_{1.9}$	-	$6.3_{0.2}$	$61.2_{0.6}$
	R2L	$\mathbf{89.6_{1.8}}$	$76.2_{1.4}$	-	$14.3_{1.0}$	$70.1_{1.1}$
	Char.	$9.5_{0.6}$	$2.3_{1.0}$	-	$0.2_{0.1}$	$0.0_{0.0}$
	Learned	$89.3_{1.6}$	$\mathbf{76.3_{0.9}}$	-	$\mathbf{23.0_{2.6}}$	$\mathbf{70.9_{0.8}}$
Acronym	Canonical	$74.3_{1.8}$	$45.0_{1.8}$	$29.5_{1.2}$	$22.1_{1.0}$	-
	Char.	$55.2_{1.6}$	$9.3_{1.9}$	$30.3_{2.5}$	$0.0_{0.0}$	-
	Learned	$\mathbf{87.8_{2.4}}$	$\mathbf{67.3_{3.4}}$	$\mathbf{45.0_{1.8}}$	$\mathbf{35.6_{3.8}}$	-
Character counting	Canonical	$36.7_{1.9}$	$32.4_{1.8}$	$39.7_{1.5}$	$\mathbf{45.2_{2.9}}$	$52.2_{2.6}$
	Char.	$31.3_{2.6}$	$29.4_{2.3}$	$43.4_{2.9}$	$41.3_{2.1}$	$52.5_{2.0}$
	Learned	$\mathbf{44.4_{4.9}}$	$\mathbf{39.1_{0.6}}$	$\mathbf{49.3_{3.9}}$	$44.1_{1.5}$	$\mathbf{58.0_{1.4}}$
Contains character	Canonical	$86.1_{2.2}$	$65.5_{0.4}$	$81.1_{1.0}$	$58.6_{0.8}$	$58.8_{0.9}$
	Char.	$65.0_{1.5}$	$64.1_{0.5}$	$80.7_{1.2}$	$38.7_{1.4}$	$52.9_{0.4}$
	Learned	$\mathbf{92.4_{5.1}}$	$\mathbf{79.5_{2.8}}$	$\mathbf{86.7_{1.3}}$	$\mathbf{69.1_{3.7}}$	$\mathbf{77.4_{1.5}}$
Character substitution	Canonical	$30.2_{0.9}$	$22.8_{0.9}$	$23.8_{1.2}$	$25.0_{2.0}$	$57.9_{2.4}$
	Char.	$26.3_{3.0}$	$18.3_{2.9}$	$33.8_{1.5}$	$13.1_{3.2}$	$47.6_{3.8}$
	Learned	$\mathbf{39.2_{3.3}}$	$\mathbf{37.0_{1.9}}$	$\mathbf{34.1_{1.0}}$	$\mathbf{30.0_{1.7}}$	$\mathbf{61.2_{4.5}}$
Character deletion	Canonical	$55.5_{1.7}$	$53.4_{2.1}$	$50.3_{3.5}$	$37.9_{1.3}$	$86.8_{2.4}$
	Char.	$53.8_{1.4}$	$27.6_{3.4}$	$57.8_{3.0}$	$30.4_{1.3}$	$74.2_{1.1}$
	Learned	$\mathbf{60.8_{2.2}}$	$\mathbf{61.7_{2.1}}$	$\mathbf{58.3_{1.7}}$	$\mathbf{45.2_{1.3}}$	$\mathbf{87.0_{3.7}}$
Character insertion	Canonical	$3.2_{0.2}$	$\mathbf{10.1_{1.3}}$	$4.3_{0.9}$	$6.5_{0.7}$	$11.7_{1.5}$
	Char.	$9.0_{1.0}$	$4.9_{1.0}$	$2.2_{0.2}$	$3.3_{0.5}$	$12.9_{0.7}$
	Learned	$\mathbf{11.7_{0.9}}$	$9.9_{2.0}$	$\mathbf{5.7_{2.2}}$	$\mathbf{8.2_{1.6}}$	$\mathbf{20.0_{2.0}}$
Code description	Canonical	$64.9_{1.9}$	$\mathbf{83.6_{0.9}}$	-	$\mathbf{75.5_{0.3}}$	-
	Char.	$82.2_{2.0}$	$49.1_{2.6}$	-	$32.2_{2.7}$	-
	Learned	$\mathbf{88.8_{2.4}}$	$81.7_{0.6}$	-	$72.9_{1.9}$	-

four in-context demonstrations from each prompt to evaluate zero-shot performance and split the examples 60/40 between training and evaluation.

Code understanding. We use the Python code-description task of Zheng et al. (2025a), based on XLCoST snippets (Zhu et al., 2022). The model selects the correct description of a snippet from four candidates. This task extends evaluation to longer structured inputs with a wider range of lengths. We split the snippets 60/40 between training and evaluation.

Models. We evaluate our method on four open-weight instruction-tuned models, Llama-3.1-8B-Instruct, Llama-3.2-3B-Instruct, Qwen-2.5-7B-Instruct, and OLMo-2-7B-Instruct (Grattafiori et al., 2024; Meta, 2024; Qwen et al., 2025; Groeneveld et al., 2024), and the closed-weight GPT-3.5-turbo-instruct (OpenAI). All models remain frozen. In every setting, retokenizer training uses only generated responses and the binary rewards computed on them, with no access to model parameters or hidden states.

We exclude Qwen-2.5-7B-Instruct from arithmetic because its single-digit number tokens leave each operand with only one valid tokenization. We also omit settings with canonical performance already near saturation ($> 95\%$ accuracy), leaving little measurable headroom for improvement: GPT-3.5-turbo-instruct on acronym generation and code description, and Qwen-2.5-7B-Instruct on code description.

3.2 Evaluation and Baselines

We train a separate retokenizer for each task–model pair. For each example in a training batch, we sample multiple retokenizations $\boldsymbol{z} \sim \pi_\theta(\cdot \mid \boldsymbol{x})$ at temperature 1, generate responses from the frozen language model, and verify them to obtain binary rewards for updating the retokenizer. At test time, we greedily decode one retokenization per held-out input and evaluate the frozen model on the resulting token sequence. We repeat each experiment over three seeds, varying both retokenizer initialization and sampled training and test examples. Table 1 reports mean accuracy and standard deviation.

Alongside canonical tokenization, we evaluate two non-learned baselines from prior work.

Character-level tokenization. We split the input into individual characters using tokens from each model's existing vocabulary. This exposes character identities directly, which has been shown to improve code understanding and character-manipulation performance (Zheng et al., 2025a).[2]

Right-to-left (R2L) tokenization. For multi-digit arithmetic, we also evaluate R2L tokenization, which has been shown to improve arithmetic accuracy (Singh & Strouse, 2024; Zheng et al., 2025a). All models evaluated on this task have tokens for every three-digit sequence. Their canonical tokenizers group digits in threes from the most significant side, whereas R2L groups them from the least significant side, aligning token boundaries with place values across operands of different lengths. For example, `1234567` becomes `[123][456][7]` under canonical tokenization and `[1][234][567]` under R2L.

3.3 Results

As shown in Table 1, learned retokenization improves over canonical tokenization in 32 of the 36 evaluated model–task pairs, with an average gain of 9.4% points, and it outperforms or closely matches the relevant hand-designed baseline. The same pattern holds for the closed-weight GPT-3.5-turbo-instruct model, where learned retokenization improves over canonical tokenization on all six evaluated tasks.

For arithmetic, R2L provides a strong hand-designed baseline, improving over canonical tokenization across all four evaluated models. For instance, the gain is nearly 40% for Llama3-8B. Learned retokenization closely matches the improved accuracy of R2L for both Llama models and GPT-3.5-turbo-instruct, while outperforming it by approximately 9% for OLMo-2-7B.

For the remaining tasks, character-level tokenization is much less consistent. On code understanding, for example, it improves Llama-8B accuracy by approximately 17% but reduces Llama-3B accuracy by approximately 35%. Learned retokenization instead outperforms character-level tokenization in every setting. This contrast highlights the advantage of learning: hand-designed strategies can improve performance when the task suggests a clear rule, as in arithmetic, but may not be optimal for every model. Learning the rules from the downstream performance removes the need to design such rules per task and per model in advance.

4 Analyzing the Learned Tokenizations

We analyze how learned tokenization strategies differ across models (Section 4.1) and whether their performance gains can be explained by increased inference cost (Section 4.2). These analyses examine both the learned token boundaries and the benefit of processing longer inputs.

4.1 The Retokenizer Learns Model-Dependent Strategies

We focus on arithmetic because, unlike most other tasks, its learned tokenizations exhibit recurring patterns that can be summarized and compared across models. Moreover, R2L provides a strong,

[2]For character-level tasks, we also evaluate a targeted variant that splits only the target word into characters and leaves the rest of the prompt canonically tokenized. Its results are similarly mixed across tasks and models; see Appendix C.

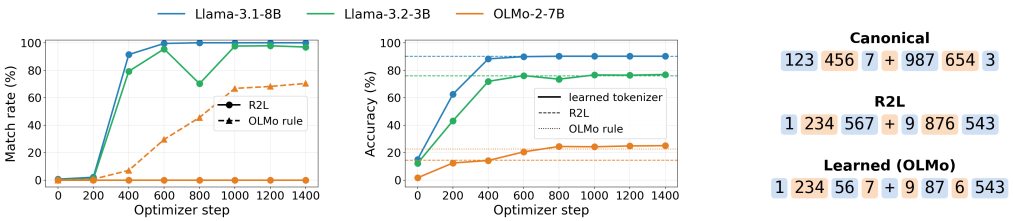

Figure 3: **The retokenizer learns different strategies across models for the same task. (a)** Fraction of retokenized test samples matching R2L during training; for OLMo, we additionally track the most common pattern learned at the final checkpoint. **(b)** Test accuracy during training, with horizontal lines showing the accuracies of the corresponding fixed tokenization rules, linking the emergence of these patterns to performance gains. **(c)** Example canonical and R2L tokenizations, together with the most common rule learned for OLMo at the final checkpoint, which uses a different grouping of digits across the two operands. See Sec. 4.1.

task-motivated alternative to canonical tokenization (Section 3.3), allowing us to ask whether the retokenizer discovers this intuitive rule from task feedback.

For both Llama models, the retokenizer progressively converges to R2L, as shown in Figure 3. By the end of training, nearly every test expression follows this rule, even though canonical tokenization already produces correct answers for roughly half of the examples. As R2L emerges, accuracy also increases and eventually reaches the R2L baseline.

The retokenizer trained with OLMo behaves differently. Although R2L improves this model's accuracy over canonical tokenization, the retokenizer never applies it during training. Instead, it progressively adopts an asymmetric digit-grouping pattern, which covers approximately 70% of test expressions by the final checkpoint (Figure 3-(c)). When applied as a fixed rule to all test expressions, this pattern achieves approximately 22% accuracy, compared with 14% for R2L. Its advantage over R2L is therefore comparable to the gain that R2L itself provides over canonical tokenization.

These results suggest that effective tokenization depends on both the task and the model. While the Llama models benefit from the intuitive R2L rule, OLMo benefits more from a different grouping of digits. Learning from downstream feedback can recover either strategy without requiring a rule to be specified for each model in advance.

4.2 Performance Gains versus Inference Cost

Beyond arithmetic, learned token boundaries are harder to summarize, but they share a clear tendency toward finer-grained tokenization. For example, on code understanding, Llama-8B receives an average of 594 tokens under learned retokenization, compared with 274 under canonical tokenization. The model therefore processes more tokens to solve the same task, raising the question of whether the gains reflect learned boundaries or simply additional inference-time computation.

We examine this question by comparing learned retokenization with four non-learned alternatives that process comparable or greater numbers of input tokens. Figure 4 plots each method's accuracy gain over canonical tokenization against its relative input token count for each model–task pair. We use input token count as a proxy for inference cost, summed across the five responses for majority voting. The baselines test distinct ways of spending this additional computation:

- **Random tokenization** (red) replaces the retokenizer's learned next-token probabilities with a uniform distribution over permitted tokens. This tests whether similarly long sequences help without learned boundary preferences.
- **Character-level tokenization** (orange) splits the input into individual characters. It tests whether maximal granularity alone explains the gains.
- **Probabilistic majority vote** (purple) samples five probabilistic tokenizations, generates a response for each, and selects the majority answer (Sathe et al., 2025). It tests whether aggregating responses across non-learned tokenizations provides comparable gains.

- **Prompt repetition** (yellow) repeats the canonically tokenized prompt. It tests whether processing the same content twice helps without changing token boundaries, and is considered a simple trick for improving performance on several tasks (Leviathan et al., 2025).

Learned retokenization uses approximately $1.8\times$ as many input tokens as canonical tokenization and improves accuracy by nearly 10 percentage points on average. Random tokenization produces sequences of similar average length to learned retokenization, but decreases accuracy for most model–task pairs and on average by approximately 7% points. Character-level tokenization processes roughly $4\times$ as many tokens as canonical tokenization, improves performance in some settings, but on average lowers accuracy by 11 points. Prompt repetition and probabilistic majority vote each improve accuracy by only about 2 points, while processing $2\times$ and nearly $6\times$ as many input tokens, respectively. These comparisons indicate that simply increasing the number of processed tokens is not sufficient to recover the gains of learned retokenization; the choice of token boundaries matters beyond the added inference-time computation.

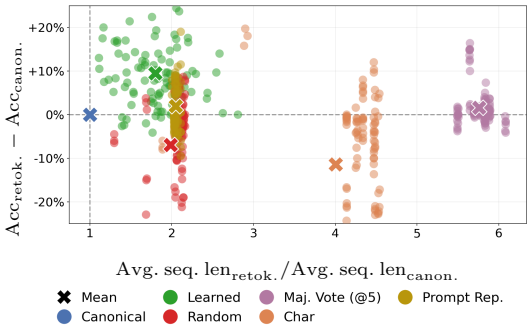

Figure 4: **Accuracy gains versus added inference cost from longer tokenized inputs.** Each point is a model–task pair under learned retokenization or one of four alternatives, showing accuracy gain over canonical tokenization versus average input sequence length normalized by canonical length. The upper-left is preferable: larger gains with less added computation, and learned retokenization lies closest to this region. See Sec. 4.2.

5 Related Work

Tokenization matters. Subword tokenization can limit model performance, leading to poor handling of morphological structure (Hofmann et al., 2022), differences in downstream performance across languages (Rust et al., 2021), and sensitivity to spelling and formatting variations (Chai et al., 2024). Its effects are particularly visible in numerical and character-level reasoning: tokenization influences numerical representations and arithmetic accuracy (Wallace et al., 2019; Singh & Strouse, 2024; Zheng et al., 2025a), while models struggle with tasks that require operating directly on individual characters (Zhang & He, 2024; Cosma et al., 2025). Recent work has therefore argued for treating tokenizer design as an important component of language-model development rather than a fixed preprocessing choice (Alqahtani et al., 2026).

Training and post-training with non-canonical tokenizations While models can be fine-tuned to perform better under canonical tokenization (e.g., Zhang & He, 2024), other works address tokenization issues more directly by changing the training procedure, model architecture, or tokenizer. Originally explored in machine translation and later extended to language models, several methods train models on randomly sampled tokenizations rather than only the canonical one, improving model performance and robustness (Kudo, 2018; Provilkov et al., 2020; Steger et al., 2026). Other approaches avoid fixed subword tokenization through character- or byte-level models (Clark et al., 2022; Xue et al., 2022), or learn the token boundaries together with the language model (Owodunni et al., 2026; Dauncey & Wattenhofer, 2026). Zheng et al. (2024) instead adapt the tokenizer itself during language-model training, iteratively refining its vocabulary based on the model's perplexity.

Closer in spirit to our work, Hiraoka et al. (2020) learn task-specific tokenizations from downstream loss in more classical NLP settings, jointly training the model and tokenization. Finally, some methods adapt the tokenizer for a given downstream task by building a task- or domain-specific vocabulary and learning embeddings for the new tokens (Liyanage & Yvon, 2026; Feher et al., 2025). Unlike all these methods, we keep both the model and its vocabulary fixed, and only change how the input string is split into tokens using the existing vocabulary.

Non-canonical tokenization at inference time. Changing only how an input is tokenized can alter the behavior of a frozen language model, even when the underlying string is unchanged. This

has been shown particularly in adversarial settings: carefully chosen non-canonical tokenizations can bypass model safety restrictions (Geh et al., 2025), while randomizing the tokenization can disrupt adversarial attacks relying on adding suffixes to the input (Jain et al., 2023). Beyond adversarial settings, hand-designed tokenization rules have improved performance on text classification, arithmetic, character manipulation, and code understanding (Hofmann et al., 2022; Singh & Strouse, 2024; Zheng et al., 2025a). However, these rules are designed for particular tasks and, as shown in Table 1, may not consistently benefit every model. Rather than choosing a single rule, Sathe et al. (2025) show that taking a majority vote over the outputs produced from many randomly sampled non-canonical tokenizations can improve model performance, at the cost of a substantial increase in inference-time computation. We instead learn a task- and model-specific tokenization strategy from downstream examples and use a single retokenization at test time.

Reinforcement learning for discrete prompt optimization Methodologically, our work is similar to prior methods that use reinforcement learning to train a small policy network to adjust the prompt, using downstream rewards computed after querying a frozen language model (Deng et al., 2022; Jafari et al., 2024; Zhang et al., 2023; Li et al., 2026). These methods optimize the model's input by adding, editing, or highlighting prompt content. Our retokenizer is subject to a stricter constraint: it cannot change the input string or vocabulary, and can only change the boundaries between its tokens. This lets us isolate how much downstream performance is limited by canonical tokenization and can be recovered by choosing a different valid tokenization of the same input.

6 Discussion and Limitations

Our experiments provide evidence that some failures of language models arise not because the required capability is entirely absent, but because canonical tokenization does not expose the input in a form that the model can operate on effectively. In such cases, performance can be improved without fine-tuning the model or changing its vocabulary, by using a non-canonical tokenization. We therefore propose learning an effective retokenization method from the downstream task by navigating the pretrained model's tokenization space.

The tokenization strategies that work best are also not universal. They can depend on both the task and the particular language model, and outside controlled settings such as arithmetic they are often difficult to summarize with a simple human-interpretable rule. This points toward a broader role for adaptive or dynamic tokenization, where the segmentation can depend on the input, task, or model rather than following a fixed rule in all settings. Our approach is one way to realize this idea while avoiding costly adaptations of the language model or its vocabulary.

Finally, we note that retokenization cannot supply knowledge or reasoning that the frozen model does not already have. Even when the model does have some competence on a given task, improvement is not guaranteed, since its errors may reflect a genuine capability limit of the model rather than a tokenization issue. Predicting in advance which tasks have recoverable performance with alternative tokenizations remains an important open question.

AI use statement

We primarily used generative AI tools as coding assistants for execution and debugging experimental code and producing visualizations. The AI-assisted code was tested for correctness by one author. Generative AI tools were also used to improve the clarity and readability of author-written text and, to a more limited extent, to provide feedback on the experimental methodology and searching for potential overlooked literature. The AI-assisted outputs were reviewed and verified by at least one author.

Generative AI tools were not used to determine the research questions, develop the central method, formulate or refine hypotheses, or interpret the experimental results. Uses involving synthetic data generation, theoretical or mathematical claims and proofs, translation, and qualitative or thematic analysis were not applicable to this work. We take responsibility for the final content of this work, including text, code, and other artifacts produced with the aid of generative AI.

REFERENCES

Sawsan Alqahtani, Mir Tafseer Nayeem, Md Tahmid Rahman Laskar, Tasnim Mohiuddin, and M Saiful Bari. Stop taking tokenizers for granted: They are core design decisions in large language models. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8410–8432, Rabat, Morocco, March 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.394. URL https://aclanthology.org/2026.eacl-long.394/.

Yekun Chai, Yewei Fang, Qiwei Peng, and Xuhong Li. Tokenization falling short: On subword robustness in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 1582–1599, 2024.

Jonathan H Clark, Dan Garrette, Iulia Turc, and John Wieting. CANINE: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91, 2022.

Adrian Cosma, Stefan Ruseti, Emilian Radoi, and Mihai Dascalu. The strawberry problem: Emergence of character-level understanding in tokenized language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 28240–28251, 2025.

Sam Dauncey and Roger Wattenhofer. You can learn tokenization end-to-end with reinforcement learning. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0rXBwsTWDB.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. RLPrompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3369–3391, 2022.

Lukas Edman, Helmut Schmid, and Alexander Fraser. CUTE: Measuring LLMs' understanding of their tokens. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3017–3026, 2024.

Darius Feher, Ivan Vulić, and Benjamin Minixhofer. Retrofitting large language models with dynamic tokenization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 29866–29883, 2025.

Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.

Renato Geh, Zilei Shao, and Guy Van den Broeck. Adversarial tokenization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 20738–20765, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, William Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah Smith, and Hannaneh Hajishirzi. OLMo: Accelerating the science of language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15789–15809, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.841. URL https://aclanthology.org/2024.acl-long.841/.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, et al. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*, 2025.

Tatsuya Hiraoka, Sho Takase, Kei Uchiumi, Atsushi Keyaki, and Naoaki Okazaki. Optimizing word segmentation for downstream task. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1341–1351, 2020.

Valentin Hofmann, Hinrich Schuetze, and Janet Pierrehumbert. An embarrassingly simple method to mitigate undesirable properties of pretrained language model tokenizers. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 385–393, 2022.

Yasaman Jafari, Dheeraj Mekala, Rose Yu, and Taylor Berg-Kirkpatrick. MORL-prompt: An empirical analysis of multi-objective reinforcement learning for discrete prompt optimization. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9878–9889, 2024.

Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping-yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. Baseline defenses for adversarial attacks against aligned language models. *arXiv preprint arXiv:2309.00614*, 2023.

Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 66–75, 2018.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Prompt repetition improves non-reasoning LLMs. *arXiv preprint arXiv:2512.14982*, 2025.

Shaoang Li, Yanhang Shi, Yufei Li, Mingfu Liang, Xiaohan Wei, Yunchen Pu, Fei Tian, Chonglin Sun, Frank Shyu, Luke Simon, et al. Learning evidence highlighting for frozen LLMs. *arXiv preprint arXiv:2604.22565*, 2026.

Siyang Liu, Naihao Deng, Sahand Sabour, Yilin Jia, Minlie Huang, and Rada Mihalcea. Task-adaptive tokenization: Enhancing long-form text generation efficacy in mental health and beyond. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 15264–15281, 2023.

Vijini Pilana Liyanage and François Yvon. AdaptBPE: From general purpose to specialized tokenizers. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2607–2620, 2026.

Meta. Llama 3.2 model card, 2024. URL `https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md`.

OpenAI. GPT-3.5 Turbo Instruct model. `https://developers.openai.com/api/docs/models/gpt-3.5-turbo-instruct`. Accessed: September 2026.

Abraham Toluwase Owodunni, Orevaoghene Ahia, and Sachin Kumar. FLEXITOKENS: Flexible tokenization for evolving language models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17170–17190, San Diego, California, United States, July 2026. doi: 10.18653/v1/2026.findings-acl.848. URL `https://aclanthology.org/2026.findings-acl.848/`.

Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. BPE-Dropout: Simple and effective subword regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 1882–1892, 2020.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL `https://arxiv.org/abs/2412.15115`.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. How good is your tokenizer? on the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3118–3135, 2021.

Ashutosh Sathe, Divyanshu Aggarwal, and Sunayana Sitaram. Improving consistency in LLM inference using probabilistic tokenization. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 4781–4793, April 2025. doi: 10.18653/v1/2025.findings-naacl.268. URL https://aclanthology.org/2025.findings-naacl.268/.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 1715–1725, 2016.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Aaditya K Singh and DJ Strouse. Tokenization counts: the impact of tokenization on arithmetic in frontier llms. *arXiv preprint arXiv:2402.14903*, 2024.

Sophie Steger, Rui Li, Sofiane Ennadir, Anya Sims, Arno Solin, Franz Pernkopf, and Martin Trapp. Stochasticity in tokenisation improves robustness. *arXiv preprint arXiv:2604.16037*, 2026.

Eric Wallace, Yizhong Wang, Sujian Li, Sameer Singh, and Matt Gardner. Do NLP models know numbers? probing numeracy in embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5307–5315, 2019.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022.

Tianjun Zhang, Xuezhi Wang, Denny Zhou, Dale Schuurmans, and Joseph E. Gonzalez. TEMPERA: Test-time prompt editing via reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=gSHyqBijPFO.

Yidan Zhang and Zhenan He. Large language models can not perform well in understanding and manipulating natural language at both character and word levels? In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 11826–11842, 2024.

Brian Siyuan Zheng, Alisa Liu, Orevaoghene Ahia, Jonathan Hayase, Yejin Choi, and Noah A. Smith. Broken tokens? your language model can secretly handle non-canonical tokenizations. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a. URL https://openreview.net/forum?id=WrYWolqKh3.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization, 2025b. URL https://arxiv.org/abs/2507.18071.

Mengyu Zheng, Hanting Chen, Tianyu Guo, Chong Zhu, Binfan Zheng, Chang Xu, and Yunhe Wang. Enhancing large language models through adaptive tokenizers. *Advances in Neural Information Processing Systems*, 37:113545–113568, 2024.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. XLCost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.

Arithmetic

System prompt
```
You are a helpful assistant.
```
User prompt
```
5540414 + 7807609 =
```
Expected answer
```
13348023
```

Character manipulation (CUTE)

System prompt
```
You are a helpful assistant. Answer concisely and place
your answer between double quotes exactly as shown in the
prompt.
```
User prompt
```
Substitute the first specified letter with the second
specified letter in a given word.

Question: Substitute "i" with "y" in "required".
```
Assistant prefix
```
Answer: "
```
Expected answer
```
requyred
```

Code understanding

System prompt
```
You are a programming assistant trained to analyze and
interpret code snippets. When provided with a code snippet
and a set of answer choices (A, B, C, or D), your task is
to evaluate the code, determine its behavior, and select
the answer that best describes this behavior. Your
response must be a single letter: A, B, C, or D. Do not
provide explanations or additional text unless explicitly
requested.
```
User prompt
```
What does the following code do: ```def avg_of_odd_num ( n
 ) :
      return n
  n = 8
 print ( avg_of_odd_num ( n ) )
``` A. Sort an array according to count of set bits     B.
 Sort a Rotated Sorted Array     C. Average of first n odd
 naturals numbers     D. Program to find transpose of a
matrix
```
Expected answer
```
C
```

Figure 5: **Task examples and prompt formats.** Examples of the input and expected output used for arithmetic, character-manipulation, and code-understanding tasks, along with the corresponding system prompts.

A Experimental Details

Retokenizer module We implement the retokenizer as a two-layer decoder-only Transformer with 8 attention heads, hidden dimension 512, and absolute positional embeddings. A special `[RETOK]` token separates the canonically tokenized input from the tokens generated by the retokenizer.

Optimization We use GSPO (Zheng et al., 2025b) to optimize the retokenizer. For each input, we sample five tokenizations from the current policy with temperature 1, evaluate them using the frozen language model, and reuse the resulting samples for three GSPO update before collecting a new batch. This reduces the number of expensive language-model calls and improves sample efficiency. We use a batch size of 32, except for code understanding, where we use 16. We train for 3 epochs on arithmetic on all models except GPT-3.5-turbo-instruct, for which we use 6. For all the remaining task-models, we train for 10 epochs. The LLM's responses during training and evaluation are generated with greedy decoding.

Adaptive entropy regularization. In some settings, we find that the retokenizer can become overly confident on a small set of tokenizations early in training. When this occurs, to encourage exploration, we add an entropy regularization to the objective function. Specifically, we use the adaptive entropy control of He et al. (2025) for GPT-3.5-turbo-instruct on the arithmetic task as without the regularization the retokenizer gets stuck at the canonical tokenization early in training. We do not use entropy regularization in the remaining experiments.

Data and prompt formatting. Figure 5 shows examples of the input format and system prompt used for each task. For each language model, we then apply its default instruction template, while retokenizing only the user prompt and keeping the remaining template tokens canonically tokenized.

B Retokenization Examples

For tasks beyond arithmetic, the learned tokenizations generally do not reduce to a simple rule shared across examples. In Figures 6–8 we show examples, where the learned retokenization corrects the model's prediction when canonical gets it wrong, on a few task and model examples.
```

Canonical

```
Come up with a sequence of words where the first letters would form this acronym: mtolv
```
Output: `Majestic Trees Often Vary`

```
Come up with a sequence of words where the first letters would form this acronym: maire
```
Output: `Majestic Animals Roam Eternally`

```
Come up with a sequence of words where the first letters would form this acronym: niusz
```
Output: `Nuclear Institute Uniting Scientists Worldwide`

Learned

```
Come up with a sequence of words where the first letters would form this acronym: mtolv
```
Output: `Modern Technology Opens Large Ventures`

```
Come up with a sequence of words where the first letters would form this acronym: maire
```
Output: `Majority Always Inspires Remarkable Effort`

```
Come up with a sequence of words where the first letters would form this acronym: niusz
```
Output: `New Innovative Uniting Systems Zone`

Figure 6: **Acronym generation with Llama-3-8B.** Canonical versus learned tokenization.

Canonical

```
What does the following code do: ```def Add ( x , y ) :
    if ( y == 0 ) :
    return x
  else :
    return Add ( x ^ y , ( x & y ) << 1 )
  ``` A. Length of direct common tangent between the two non    B. Distance between a point and a Plane in 3 D    C. Add two numbers without using arithmetic operators    D. Minimize the total number of teddies to be distributed
```
Output: `A`

Learned

```
What does the following code do: ```def Add ( x , y ) :
    if ( y == 0 ) :
    return x
  else :
    return Add ( x ^ y , ( x & y ) << 1 )
  ``` A. Length of direct common tangent between the two non    B. Distance between a point and a Plane in 3 D    C. Add two numbers without using arith
metic operators    D. Minimize the total number of teddies to be distributed
```
Output: `C`

Figure 7: **Code understanding with Llama-3-8B.** Canonical versus learned tokenization.

C Additional Baseline comparison

Character-level tokenization splits the entire prompt into individual characters, even when only a small part of the input requires character-level access. We therefore consider a more targeted baseline, *partial character-level tokenization*, which splits only the target word into individual characters while keeping the rest of the prompt canonically tokenized. For example, for the prompt `Delete every instance of ``p'' in ``europe''`, partial character-level tokenization splits only `europe` into individual-character tokens, while the rest remains canonically tokenized. As shown in Table 2, this baseline is sometimes stronger than full character-level tokenization, but like full character-level tokenization, it does not consistently improve over canonical tokenization. Learned retokenization remains stronger than this more targeted heuristic in most settings, highlighting once more the benefit of learning task- and model-specific token boundaries rather than relying on a fixed heuristic.

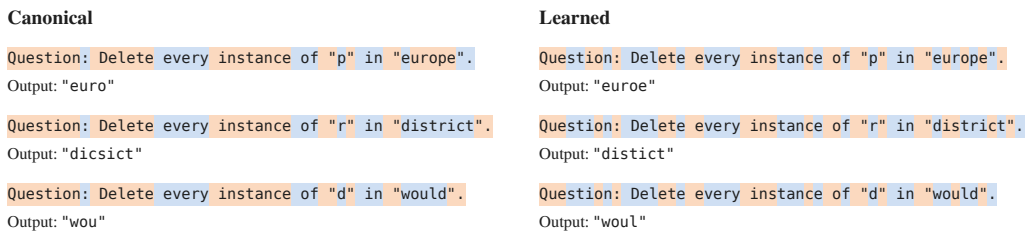

Figure 8: **Character deletion with Qwen2.5-7B.** Canonical versus learned tokenization.

Table 2: **Comparison with partial character-level tokenization.** Using the same setup as Table 1, we focus on the character-level tasks and compare learned retokenization with *partial* character-level tokenization, which splits only the target word into individual characters. Learned retokenization is stronger in most model–task settings.

Task	Method	Llama-3-8B	Llama-3-3B	Qwen2.5-7B	OLMo-2-7B	GPT-3.5-turbo
Acronym	Canonical	$74.3_{1.8}$	$45.0_{1.8}$	$29.5_{1.2}$	$22.1_{1.0}$	-
	Partial char.	$87.7_{1.9}$	$63.1_{0.1}$	$42.4_{1.6}$	$32.1_{0.4}$	-
	Learned	$\mathbf{87.8}_{2.4}$	$\mathbf{67.3}_{3.4}$	$\mathbf{45.0}_{1.8}$	$\mathbf{35.6}_{3.8}$	-
Counting characters	Canonical	$36.7_{1.9}$	$32.4_{1.8}$	$39.7_{1.5}$	$\mathbf{45.2}_{2.9}$	$52.2_{2.6}$
	Partial char.	$32.3_{3.1}$	$16.5_{1.6}$	$41.9_{2.7}$	$43.2_{2.8}$	$46.1_{2.5}$
	Learned	$\mathbf{44.4}_{4.9}$	$\mathbf{39.1}_{0.6}$	$\mathbf{49.3}_{3.9}$	$44.1_{1.5}$	$\mathbf{58.0}_{1.4}$
Contains character	Canonical	$86.1_{2.2}$	$65.5_{0.4}$	$81.1_{1.0}$	$58.6_{0.8}$	$58.8_{0.9}$
	Partial char.	$\mathbf{97.6}_{0.5}$	$70.9_{2.5}$	$84.0_{1.0}$	$41.8_{3.7}$	$58.1_{0.5}$
	Learned	$92.4_{5.1}$	$\mathbf{79.5}_{2.8}$	$\mathbf{86.7}_{1.3}$	$\mathbf{69.1}_{3.7}$	$\mathbf{77.4}_{1.5}$
Character substitution	Canonical	$30.2_{0.9}$	$22.8_{0.9}$	$23.8_{1.2}$	$25.0_{2.0}$	$57.9_{2.4}$
	Partial char.	$31.3_{1.2}$	$24.4_{4.2}$	$\mathbf{35.7}_{3.7}$	$28.0_{2.8}$	$\mathbf{62.2}_{3.4}$
	Learned	$\mathbf{39.2}_{3.3}$	$\mathbf{37.0}_{1.9}$	$34.1_{1.0}$	$\mathbf{30.0}_{1.7}$	$61.2_{4.5}$
Character deletion	Canonical	$55.5_{1.7}$	$53.4_{2.1}$	$50.3_{3.5}$	$37.9_{1.3}$	$86.8_{2.4}$
	Partial char.	$60.2_{1.9}$	$47.6_{1.0}$	$\mathbf{59.7}_{2.1}$	$42.6_{2.5}$	$86.1_{1.3}$
	Learned	$\mathbf{60.8}_{2.2}$	$\mathbf{61.7}_{2.1}$	$58.3_{1.7}$	$\mathbf{45.2}_{1.3}$	$\mathbf{87.0}_{3.7}$
Character insertion	Canonical	$3.2_{0.2}$	$\mathbf{10.1}_{1.3}$	$4.3_{0.9}$	$6.5_{0.7}$	$11.7_{1.5}$
	Partial char.	$1.7_{0.3}$	$6.7_{0.7}$	$2.6_{0.5}$	$6.5_{0.7}$	$15.1_{1.5}$
	Learned	$\mathbf{11.7}_{0.9}$	$9.9_{2.0}$	$\mathbf{5.7}_{2.2}$	$\mathbf{8.2}_{1.6}$	$\mathbf{20.0}_{2.0}$