Compressing Sequences in the Latent Embedding Space: K-Token Merging for Large Language Models

Zihao Xu[♢*], **John Harvill**[♡*], **Ziwei Fan**[♠], **Yizhou Sun**[♣†], **Hao Ding**[♠], **Hao Wang**[♢△],
[♢]Rutgers University, [♠]AWS AI Labs, [♣]Amazon, [♡]Mistral AI
[†]University of California Los Angeles, [△]University of Illinois Urbana-Champaign
zihao.xu@rutgers.edu

Abstract

Large Language Models (LLMs) incur significant computational and memory costs when processing long prompts, as full self-attention scales quadratically with input length. Token compression aims to address this challenge by reducing the number of tokens representing inputs. However, existing prompt-compression approaches primarily operate in token space and overlook inefficiencies in the **latent embedding space**. In this paper, we propose K-Token Merging, a latent-space compression framework that merges each contiguous block of K token embeddings into a single embedding via a lightweight encoder. The compressed sequence is processed by a LoRA-adapted LLM, while generation remains in the original vocabulary. Experiments on structural reasoning (Textualized Tree), sentiment classification (Amazon Reviews), and code editing (CommitPackFT) show that K-Token Merging lies on the Pareto frontier of performance vs. compression, achieving up to **75%** input length reduction with minimal performance degradation. On a Qwen Coder 7B model, 2-Token Merging reduces latency by 20.9% and improves throughput by 27.8% with only a 1.1% relative increase in perplexity. Code is available at https://github.com/shsjxzh/K-Token-Merging.

1 Introduction

We have witnessed tremendous progress in Large Language Models (LLMs) (Zhao et al., 2023). Since the advent of ChatGPT (Brown et al., 2020), LLMs have become deeply integrated into many aspects of daily life – from AI-enhanced customer support (Shareef, 2024; Scatolin and Pedrini, 2026) and game NPCs (Cox and Ooi, 2023; Christiansen et al., 2024) to agentic systems capable of data analysis (Tang et al., 2025) and complex software generation (Joel et al., 2024; Jiang et al., 2026).

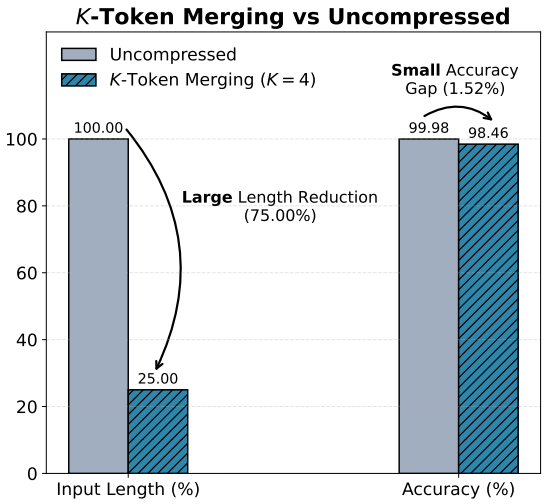


Figure 1: Our K-Token Merging method ($K = 4$) achieves a 75% reduction in input length with only a 1.52% drop in accuracy on the Textualized Tree benchmark, demonstrating that it exploits redundancy in the latent embedding space while preserving high performance. See the "Experiments" section for details.

These increasingly sophisticated applications demand ever-longer input contexts, which sharply escalate both computational and memory costs: with softmax attention (Vaswani et al., 2017), the memory and compute requirements of LLMs grow **quadratically** with input length.

A natural direction to mitigate this challenge is *token compression* (Li et al., 2025). Token compression aims to reduce the input length by representing prompts with fewer tokens. Existing approaches largely fall into two categories.

The first category, **hard prompt compression** (Li et al., 2023; Pan et al., 2024; Jiang et al., 2024; Liskavets et al., 2025), decreases token count by dropping or summarizing tokens. While effective for tasks where key information is sparse, these methods often fail on information-dense tasks such as document revision, mathematical reasoning, symbolic computation, or code translation–settings in which every token may carry essential

[*]Work done at AWS AI Labs.

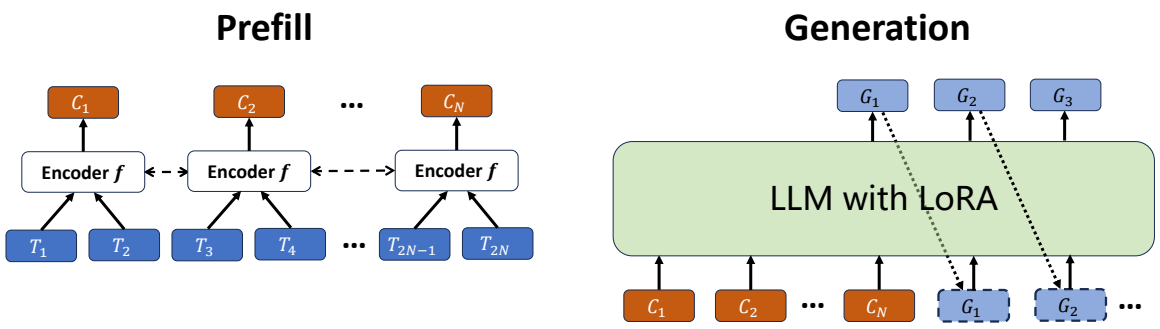


Figure 2: **Model Structure for K-Token Merging Model (Case $K = 2$). Left:** During the prefill stage, the encoder f takes each K consecutive input tokens and produces a single compressed token embedding. Here, T_i denotes the original input tokens and C_i denotes the resulting compressed tokens. **Right:** During the generation stage, the LLM outputs *original* (uncompressed) tokens. Each newly generated token is appended to the mixed compressed/uncompressed prefix, after which standard auto-regressive generation continues. Here, G_i denotes the generated uncompressed tokens.

information.

The second category, **soft prompt compression** (Mu et al., 2023; Harvill et al., 2025; Gao et al., 2024), draws inspiration from soft prompts (Lester et al., 2021) and learns new tokens or adapts model parameters to produce more compact representations. These methods have gained increasing attention because, empirically, they preserve more information than hard methods while still offering substantial compression (Harvill et al., 2025; Gao et al., 2024). However, existing methods focus almost exclusively on reducing the number of *tokens*, overlooking a major source of redundancy: *the embedding space itself*.

Consider the QWEN 2.5 model (Qwen et al., 2024), which has a vocabulary size of 151,936. Each token consumes approximately $896 \times 32 = 28{,}672$ bits, whereas the theoretical minimum needed to identify one token from its vocabulary is only $\log_2(151{,}936) \approx 18$ bits. For a K-gram, the minimum grows to merely $\log_2(151{,}936^K) \approx 18K$ bits. This enormous gap highlights significant inefficiency in current embeddings and indicates substantial room for improving input representation.

In this paper, we propose **K-Token Merging**, a new soft prompt compression method that directly targets redundancy in the *latent embedding space*. Our key idea is to use a learned encoder to compress a sequence of K consecutive tokens (K-grams) into a single embedding. In other words, we represent multiple tokens with one embedding **on the input side only**. We then finetune the LLM with LoRA to adapt to these compressed embeddings, enabling strong downstream performance.

As illustrated in Figure 1, our approach achieves high compression ratios with minimal performance degradation (see the "Experiments" section for details).

Importantly, our method is fundamentally different from expanding the vocabulary with K-grams. Vocabulary expansion requires representing multi-token sequences as single tokens *on both the input and output sides*. This leads to an explosion of low-frequency "tail" tokens, which severely complicates optimization and becomes quickly infeasible for large K. For example, the number of unique 4-grams can exceed the size of the original vocabulary by 29× (Yu et al., 2025), resulting in prohibitive storage overhead. Furthermore, since each K-gram is assigned a fixed embedding, such approaches cannot generalize to unseen K-grams during inference, sharply limiting their applicability. In contrast, our encoder-based approach avoids tail-token explosion, supports large K values without increasing vocabulary size, and generalizes naturally to unseen token combinations.

Our main contributions can be summarized as follows:

- We introduce **K-Token Merging**, a novel framework for latent-space token sequence compression.
- We present a training recipe for adapting LLMs to such compressed inputs.
- We show experimentally that our approach achieves up to **75% token compression** with minimal performance degradation across diverse tasks. In scenarios where output length is much shorter than input length, this corre-

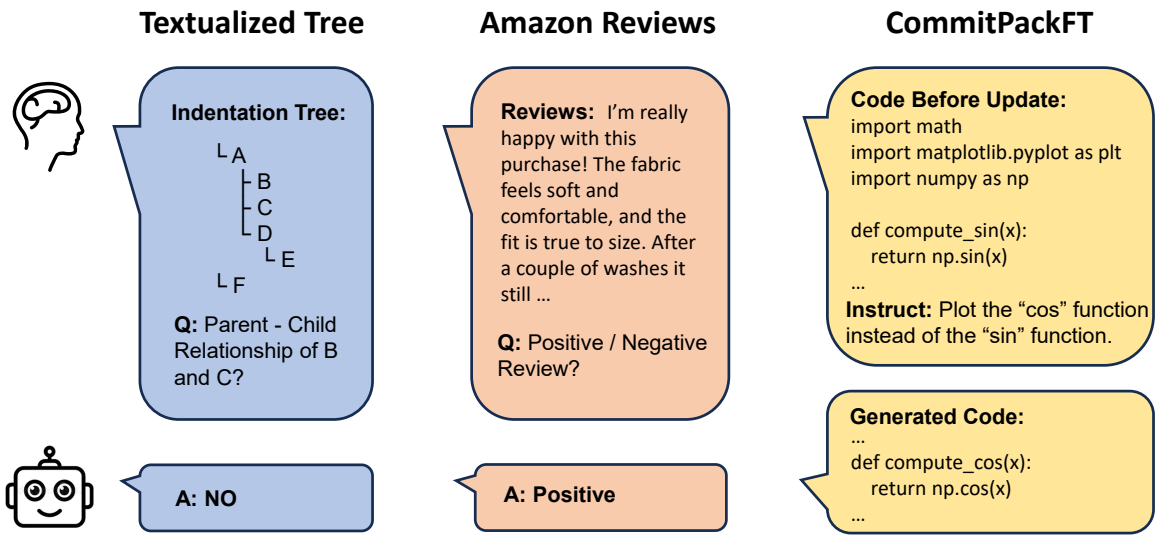


Figure 3: **Datasets & Tasks. Left: Textualized Tree.** Given a textualized indentation tree, the LLM determines whether two nodes have a parent - child relationship. **Middle: Amazon Reviews.** The LLM performs sentiment classification to judge whether a product review is positive or negative. **Right: CommitPackFT.** Given code and an update instruction, the LLM needs to output the modified code that follows the instruction.

sponds to an estimated **94%**[1] **reduction in computation**. Additional component ablations and experiments with a 7B model validate the design and demonstrate that its runtime benefits persist at a larger scale.

2 Related Work

Token compression (Li et al., 2025) has been extensively studied since the introduction of the Transformer architecture (Vaswani et al., 2017). Existing approaches can be grouped into *hard prompt compression* and *soft prompt compression*.

Hard prompt compression. Hard compression methods (Li et al., 2023; Pan et al., 2024; Chuang et al., 2024; Jiang et al., 2024; Jung and Kim, 2024; Shandilya et al., 2025; Liu et al., 2023; Liskavets et al., 2025) reduce input length by dropping or summarizing tokens. For example, SelectiveContext (Li et al., 2023) removes tokens deemed uninformative based on self-information, and LLMLingua2 (Pan et al., 2024) trains a classifier to identify redundant tokens. However, these methods often suffer significant performance degradation on information-dense tasks, as even a small number of omitted tokens may contain essential context.

In contrast, our method never removes tokens; it re-embeds all tokens into compressed embeddings. This enables substantially higher compression rates while mitigating information loss. Furthermore, our technique is compatible with hard compression and can be applied subsequently, making it complementary rather than competing.

Soft prompt compression. Soft compression methods (Bolya et al., 2022; Mu et al., 2023; Chevalier et al., 2023; Ge et al., 2023; Cheng et al., 2024; Harvill et al., 2025; Gao et al., 2024) learn new tokens or adapt LLM parameters to produce compact input representations. For instance, Gist (Mu et al., 2023) compresses instructions into a small set of meta-tokens prepended to the input, enforcing restricted attention patterns to ensure that later tokens cannot be attended beyond the meta-tokens. LTSC (Harvill et al., 2025) discovers frequently occurring token subsequences and replaces them with shorter learned meta-tokens, analogous to classical file-compression algorithms.

However, these methods operate entirely at the *token level*. In contrast, our approach performs compression directly in the *latent embedding space*, targeting inefficiencies in token embeddings themselves rather than the token sequence. This distinction allows our framework – though still a form of soft compression – to achieve a substantially more favorable balance between performance and compression ratio than prior approaches.

[1]With K=4, input length is reduced by 75%. Prefill attention cost scales as $O(n^2)$, so compressed prefill requires only $0.25^2 = 6.25\%$ of the original FLOPs. When output length $M \ll$ input length KN, total FLOPs $\approx 6.25\%$ of uncompressed, yielding $\sim 94\%$ reduction.

Table 1: Accuracy (%), Length Reduction Ratio (%), and P–L F_1 on Textualized Tree. We **bold** the best result and underline the second-best result for each metric. Our method achieves both the best and second-best results on Length Reduction Ratio and P–L F_1, while achieving the second-best Accuracy, surpassed only by the Uncompressed baseline.

Method	Uncompressed	SelectiveContext	LLMLingua2	LTSC	K-Token Merging (Ours)		
					2-Token	3-Token	4-Token
Accuracy	**99.98**$^{\pm 0.01}$	90.33$^{\pm 0.07}$	82.17[1]	99.68[1]	99.79$^{\pm 0.22}$	98.73$^{\pm 0.55}$	98.46$^{\pm 0.30}$
Length Reduction	0.0	52.5	27.0	27.1	50.0	66.7	**75.0**
P–L F_1	0.000	0.664	0.406	0.426	0.666	0.796	**0.851**

3 Method

3.1 Model Structure

Our insight is to exploit inefficiency in token-level representations. We introduce a lightweight encoder f that compresses every contiguous block of K tokens into a single compressed token embedding. The overall architecture is illustrated in Fig. 2, using the 2-Token Merging model as an example.

For our K-Token Merging model, we assume the total number of input tokens be KN. If necessary, we append padding tokens to ensure that the input length is always a multiple of K. We describe the model's workflow in two stages: the **prefill** stage and the **generation** stage.

Prefill. In the prefill stage, for each block $\{T_{iK+j}\}_{j=1}^{K}, i \in \{0, \ldots, N-1\}$, the encoder f takes as input their original embeddings from the model's embedding table Emb and outputs a single compressed embedding C_i:

$$C_i = f(Emb(T_{iK+1}), \ldots, Emb(T_{iK+K})) \,. \quad (1)$$

This compression is computationally inexpensive because: (i) the encoder is a small multi-layer perceptron (MLP) of roughly 50 MB, and (ii) compressed embeddings for frequently occurring K-grams may be cached.

Generation. During generation, the model always outputs *original* (uncompressed) tokens. These newly produced tokens are appended to the mixed compressed/uncompressed prefix and autoregressively processed by the LLM.

3.2 Objective Function

We finetune the LLM using a LoRA adapter jointly with the encoder f. Importantly, the training objective is evaluated only on positions corresponding to uncompressed (original) tokens – e.g., the generated tokens $G_1, G_2, G_3, \ldots$ in Fig. 2. Let the full sequence after inserting compressed tokens be

$$X = (C_1, \ldots C_N, G_1, \ldots G_M),$$

where M is the number of generated tokens. Let $\mathcal{U} = \{N+1, \ldots, N+M\}$ denote the index set of uncompressed token positions. The LLM defines next-token probabilities $p_\theta(\cdot \mid X_{<t})$. The training loss is the negative log-likelihood restricted to uncompressed targets:

$$\mathcal{L}(\theta, f) = -\sum_{t \in \mathcal{U}} \log p_\theta(X_t \mid X_{<t}) \,. \quad (2)$$

This objective ensures that the model learns to interpret compressed embeddings while maintaining generation quality on the original vocabulary.

3.3 Compressed Embedding Initialization

The initialization strategy for compressed embeddings has a significant impact on the convergence speed of training. Empirically, we find that initializing the embedding as the average (mean pooling) of the K original token embeddings leads to substantially faster convergence compared to random initialization.

We design a dedicated encoder architecture that naturally produces such an initialization. The encoder computes a residual combination of (i) the mean of the K original embeddings and (ii) the output of a small MLP. The MLP weights are initialized close to zero so that, at the beginning of training, its contribution is negligible. Consequently, the encoder initially outputs a compressed embedding that closely matches the average-pooled embedding of the K tokens, providing a stable and semantically grounded starting point for optimization.

The pseudo code for this *Average-Initialized Encoder* is shown in Alg. 1.

[1]Results come from Harvill et al. (2025).

4 Experiments

In this section, we evaluate our method across three tasks, analyze its components and embedding initialization strategy, and present a qualitative case study.

4.1 Baselines

We select two representative hard prompt compression methods, SelectiveContext (Li et al., 2023) and LLMLingua2 (Pan et al., 2024), along with a recent soft prompt compression approach, LTSC (Harvill et al., 2025), as our baselines. For comparison, we also report results from the uncompressed model.

4.2 Datasets & Tasks

We use three datasets, each designed for a unique task.

Textualized Tree (Harvill et al., 2025) is a synthetic dataset consisting of textualized indentation trees, as shown in Fig. 3 (left). We design a classification task on this dataset to evaluate whether our method preserves most of the information after prompt compression. Specifically, we randomly select a pair of nodes in the tree and ask the LLM to classify whether a parent-child relationship exists between them. The model outputs "true" if such a relationship exists, and "false" otherwise. Since any node pair can be selected, the compressed token embeddings must preserve complete structural information to maintain high classification accuracy. This dataset includes 2.45 million training samples and 50 thousand test samples, with each tree containing up to 150 nodes.

Amazon Reviews (Hou et al., 2024) is a dataset comprising user reviews, item metadata, and user-item interactions from Amazon. We formulate a sentiment classification task to test whether the compressed embeddings perform well on natural language, as shown in Fig. 3 (middle). Given a user review (e.g., "This legging is amazing"), the model is expected to classify the sentiment as either "positive" or "negative". Reviews with a rating above 3.5 are labeled as positive, and those below or equal to 3.5 as negative. We use the "Amazon Fashion" category, consisting of 2.45 million training samples and 50 thousand test samples.

CommitPackFT (Muennighoff et al., 2023) is a Git commit dataset covering 350 programming languages. Each sample contains the code before an update, the corresponding commit message, and the updated code. We define a code update task where the model receives the code before the update and the commit message as input, and is expected to generate the updated code, which is shown in Fig. 3 (right). We use the CommitPackFT subset, which consists of high-quality filtered data from the original CommitPack dataset. We further focus our experiments on the Python category, which includes 50.4 thousand training samples and 5 thousand test samples.

4.3 Evaluation Metrics

For each method, we report both the model performance with compressed prompts and the average length reduction ratio. For the Textualized Tree and Amazon Reviews datasets, we use classification accuracy as the performance metric, while for CommitPackFT, we use perplexity.

In practice, different methods may trade off performance and length reduction differently; a method with stronger compression may exhibit slightly lower performance, and vice versa. To quantitatively evaluate this trade-off, we propose the **Performance–Length Reduction F_1 score (P–L F_1)**, defined as the harmonic mean of performance and length reduction, analogous to the classical F_1 score.

The P–L F_1 score is computed as:

$$\text{P–L } F_1 = \frac{2PL}{P + L}, \tag{3}$$

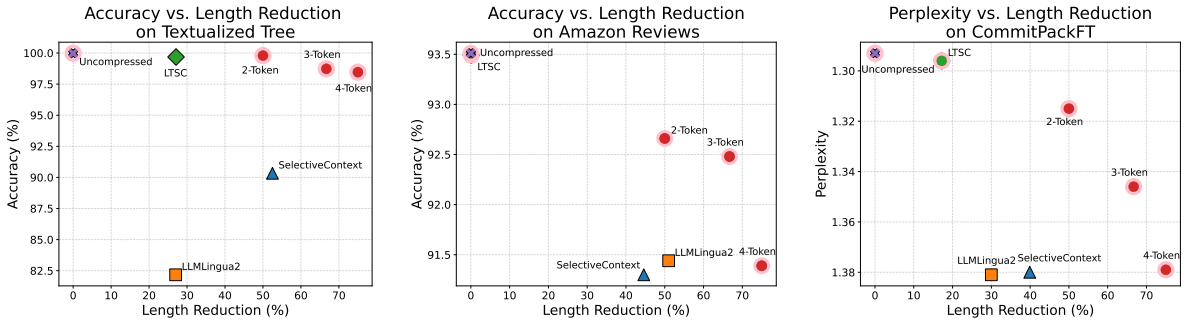
where $P \in [0, 1]$ denotes the normalized performance score and L denotes the length reduction ratio.

For the Textualized Tree and Amazon Reviews datasets, we directly use classification accuracy as P. For CommitPackFT, we use a *relative perplexity ratio*. Let ppl_i be the perplexity of method i and $\text{ppl}_{\min} = \min_i \text{ppl}_i$. The normalized performance $P_i \in [0, 1]$ for each method i is given by:

$$P_i = \frac{\text{ppl}_{\min}}{\text{ppl}_i}, \tag{4}$$

which ensures that lower perplexity corresponds to higher normalized performance.

Note that P–L F_1 is a summary metric that balances task performance and compression ratio, and may therefore favor a particular trade-off between the two. We thus use it only as supplementary evidence and base our main conclusions primarily on the raw task metrics and the Pareto frontier.



(a) Textualized Tree (b) Amazon Reviews (c) CommitPackFT

Figure 4: **Performance Score (Accuracy / Perplexity) vs. Length Reduction Ratio** on three datasets: (a) Textualized Tree, (b) Amazon Reviews, and (c) CommitPackFT. A higher Performance Score / Length Reduction Ratio indicates better performance; therefore, points located toward the upper-right region of the plot are preferred. Pareto-optimal points are marked with hollow pink circle markers (○). Our method, K-Token Merging with $K \in \{2, 3, 4\}$, is highlighted in red. As shown, our approach lies on the Pareto-optimal frontier across all three datasets. For consistency, panel (c) (CommitPackFT) uses a y-axis (perplexity) that increases from top to bottom.

Table 2: Accuracy (%), Length Reduction Ratio (%), and P–L F_1 on Amazon Reviews. We **bold** the best result and underline the second-best result for each metric. Our method achieves the best and second-best results on P–L F_1.

Method	Uncompressed	SelectiveContext	LLMLingua2	LTSC	K-Token Merging (Ours)		
					2-Token	3-Token	4-Token
Accuracy	**93.51**$^{\pm 0.03}$	91.30$^{\pm 0.01}$	91.44$^{\pm 0.12}$	93.49$^{\pm 0.05}$	92.66$^{\pm 0.18}$	92.48$^{\pm 0.22}$	91.39$^{\pm 0.39}$
Length Reduction	0.0	44.6	51.0	0.1	50.0	66.7	**75.0**
P–L F_1	0.000	0.599	0.655	0.002	0.650	0.775	**0.824**

4.4 Implementation Details

We mainly use Qwen-2.5 0.5B (Qwen et al., 2024) as the base model for all baselines. We also evaluate our method on the larger Qwen-2.5 Coder 7B (Hui et al., 2024) using CommitPackFT. We apply LoRA finetuning to this base model separately on the compressed prompts produced by different baselines, using the same LoRA configuration across all experiments (rank $r = 4$, LoRA scaling factor $\alpha = 16$, and dropout rate 0.05. We apply LoRA adaptation across all layers, including both the self-attention modules and the MLP components).

For the Tree Classification task, we adopt a curriculum learning strategy: the model is first trained on smaller trees and progresses to larger trees only after its performance surpasses a predefined threshold. This stage uses approximately 19.6–34.6 million auxiliary training samples. For all other tasks, the model is trained directly without curriculum learning.

The encoder in our model is implemented as a three-layer MLP. All models are trained using the AdamW (Loshchilov and Hutter, 2017) optimizer with a learning rate of 1×10^{-4}.

For results with repeated runs, we report the mean and sample standard deviation. Length reduction ratios and the P–L F_1 scores derived from the mean performance are reported once.

4.5 Result Discussion

Textualized Tree. Table 1 reports the classification accuracies and length reductions, which are also visualized in Fig. 4(a). Relative to the 99.98% uncompressed accuracy, 2-, 3-, and 4-Token Merging achieve mean accuracies of 99.79%, 98.73%, and 98.46% while reducing input length by 50.0%, 66.7%, and 75.0%, respectively. Thus, even the most aggressive setting loses only 1.52 percentage points of accuracy.

All three K-Token Merging configurations lie on the Pareto frontier: increasing K provides progressively greater compression with a modest accuracy tradeoff. In comparison, LLMLingua2 reaches only 82.17% accuracy at 27.0% length reduction, while LTSC preserves 99.68% accuracy but reduces length by only 27.1%. Aligned with this conclusion, the 4-Token model obtains the highest P–L F_1 score of 0.851.

Amazon Reviews. Table 2 and Fig. 4(b) show the

```
// Encoder compressing k embeddings
   into 1 embedding
class Average-Initialized Encoder:
    def init(self, embedding_dim: integer, k:
     integer):
        self.net ← MLP(input_dim = k * em-
         bedding_dim, output_dim = embed-
         ding_dim);
    def forward(self, x: matrix):
        // Input      shape:(batch_size,
           embedding_dim, k)
        mean ← Mean(x, axis = -1);
        x_flat ← Reshape(x, shape =
         (batch_size, embedding_dim * k));
        output ← mean + self.net(x_flat);
        return output;
```

Algorithm 1: Pseudo Code for Average-Initialized Encoder. Since the MLP is initialized near 0, this encoder at first outputs a compressed embedding that closely matches the average-pooled embedding of the K tokens, giving training a stable starting point.

accuracy-compression tradeoff on natural-language sentiment classification. The uncompressed model reaches 93.51% accuracy. With 2-, 3-, and 4-Token Merging, accuracy remains at 92.66%, 92.48%, and 91.39% while input length is reduced by 50.0%, 66.7%, and 75.0%, respectively. In particular, the 2-Token model loses only 0.85 percentage points while halving the input length.

Our three configurations lie on the Pareto frontier, showing that each additional level of compression yields a distinct performance–length tradeoff. LTSC nearly matches the uncompressed accuracy at 93.49% but reduces length by only 0.1%. SelectiveContext and LLMLingua2 obtain more than 40% reduction, but their accuracies remain below those of our 2- and 3-Token models. The 4-Token model obtains the highest P–L F_1 score of 0.824, indicating a favorable balance between accuracy and compression.

CommitPackFT. Table 3 and Fig. 4(c) present the perplexity–compression tradeoff on code editing, where lower perplexity is better. The uncompressed model obtains a perplexity of 1.293. The 2-, 3-, and 4-Token models obtain perplexities of 1.315, 1.346, and 1.379 while reducing input length by 50.0%, 66.7%, and 75.0%, respectively. These correspond to relative perplexity increases of only 1.7%, 4.1%, and 6.7%.

All three of our configurations lie on the Pareto frontier. By contrast, SelectiveContext and LLMLingua2 have higher perplexity than our 2-Token model while providing less length reduction, and LTSC obtains near-uncompressed perplexity (1.296) but reduces length by only 17.2%. The perplexities and Pareto dominance therefore prove that K-Token Merging offers a favorable tradeoff on code. As a result, the 4-Token model achieves the highest P–L F_1 score of 0.833.

4.6 Scaling to a Larger Model

The main experiments use a 0.5B-parameter backbone. To examine whether the runtime benefits of token merging persist at a larger scale, we additionally evaluate 2-Token Merging with a Qwen Coder 7B model on CommitPackFT. Table 4 compares it with the corresponding uncompressed model.

Relative to the uncompressed model, 2-Token Merging increases perplexity only slightly, from 1.263 to 1.277 (a 1.1% relative increase), while reducing latency by 20.9% and improving throughput by 27.8%. Peak memory usage increases by 6.6% in this setting. Due to limited computational resources, we evaluate with a batch size of only 2, under which GPU memory usage is dominated by the base model and our additional encoder, leading to the modest increase in peak memory. With larger batch sizes, however, the memory savings from token compression are expected to become more pronounced, as reducing the number of processed tokens can offset the fixed memory overhead introduced by the encoder.

4.7 Ablation Studies

Component ablation. To isolate the contribution of each component, we conduct a controlled ablation on CommitPackFT. We compare full K-Token Merging (KTM), mean pooling without the learnable encoder residual, and an uncompressed control with a matched extra-parameter budget. The last configuration uses a comparable encoder that maps K input embeddings to K output embeddings, thereby retaining the original sequence length. For this focused comparison, all configurations are trained for 10 epochs.

As shown in Table 5, adding the learnable residual improves perplexity from 1.453 to 1.447 relative to mean pooling alone, with only small differences in latency, throughput, and memory. Com-

Table 3: Perplexity, Length Reduction Ratio (%), and P–L F_1 on CommitPackFT. We **bold** the best result and underline the second-best result for each metric. On P–L F_1, our method obtains both the best and the second-best performance.

Method	Uncompressed	SelectiveContext	LLMLingua2	LTSC	K-Token Merging (Ours)		
					2-Token	3-Token	4-Token
Perplexity	**1.293**$^{\pm 0.001}$	1.380$^{\pm 0.000}$	1.381$^{\pm 0.000}$	1.296$^{\pm 0.000}$	1.315$^{\pm 0.024}$	1.346$^{\pm 0.031}$	1.379$^{\pm 0.012}$
Length Reduction	0.0	39.9	30.0	17.2	50.0	66.7	**75.0**
P–L F_1	0.000	0.560	0.454	0.293	0.663	0.787	**0.833**

Table 4: Performance and inference efficiency on the Qwen Coder 7B model. Lower perplexity, latency, and peak memory are better; higher throughput is better. The best value for each metric is bolded.

Metric	No Compression	2-Token Merge
Perplexity ↓	**1.263**	1.277
Latency (s) ↓	0.789$^{\pm 0.151}$	**0.624**$^{\pm 0.139}$
Throughput (samples/s) ↑	5.271$^{\pm 1.138}$	**6.736**$^{\pm 1.577}$
GPU peak memory (MB) ↓	**35,983**$^{\pm 1,026}$	38,349$^{\pm 938}$

Table 5: Component ablation on CommitPackFT after 10 epochs. The best value for each metric is bolded, and the second-best value is underlined.

Metric	KTM (Ours)	Mean Pooling Only	No Compression
Perplexity ↓	1.447	1.453	**1.364**
Latency (s) ↓	0.358$^{\pm 0.103}$	**0.353**$^{\pm 0.102}$	0.546$^{\pm 0.118}$
Throughput (samples/s) ↑	17.792$^{\pm 4.124}$	**18.038**$^{\pm 4.213}$	11.377$^{\pm 2.148}$
GPU peak memory (MB) ↓	9,552$^{\pm 1,314}$	**9,232**$^{\pm 1,314}$	13,006$^{\pm 1,444}$

pared with the matched uncompressed control, full KTM reduces latency by 34.4%, increases throughput by 56.4%, and lowers peak memory by 26.6%, at the cost of a slightly higher perplexity (+6.09%). These results indicate complementary roles for the two components: the learnable encoder residual recovers semantic quality beyond fixed mean pooling, while reducing the sequence length provides the primary efficiency gains.

Embedding initialization strategies ablation. As shown in Fig. 5, we present an ablation study comparing different embedding initialization strategies for the compressed embeddings. Our average-based initialization achieves 97% classification accuracy in just 8 epochs – one epoch faster than random initialization. This indicates that the average-based approach enables faster convergence and serves as a more effective initialization method than random initialization.

4.8 Case Study

We include a case study of our 2-Token Merging model trained on the CommitPackFT dataset to demonstrate that, even with compressed inputs, the model can still preserve key information and follow instructions to generate correct outputs.

Listing 1: Code input used for our model's case study.

```
import math
import matplotlib.pyplot as plt
import numpy as np

# Define the function
def compute_sin(x):
    return np.sin(x)

# Generate input values
x_vals = np.linspace(-2 * np.pi, 2 * np.
    pi, 500)
y_vals = compute_sin(x_vals)

# Plotting
plt.figure(figsize=(8, 4))
plt.plot(x_vals, y_vals, label='sin(x)')
plt.show()
```

In this example, we provide the original code in Listing 1 and instruct the model to "plot the 'cos' function at the same time."

The generated code is shown in Listing 2. Two observations are noteworthy. First, the model successfully generates an additional function to plot the cosine curve. Second, rather than introducing a new variable, it reuses the previously defined variable "x_vals" (highlighted in yellow) as the function domain, ensuring that both plots share the same range.

These two points indicate that, despite compressing both the original code and the instruction, the model can still recover most relevant information and generate the correct implementation. This suggests our method effectively leverages redundancy in the embedding space while preserving model performance.

5 Future Directions

Compression During Generation: Currently, we do not apply token compression to generated tokens for simplicity. Exploring how to extend our compression technique to the generation phase could lead to significant memory and efficiency gains.

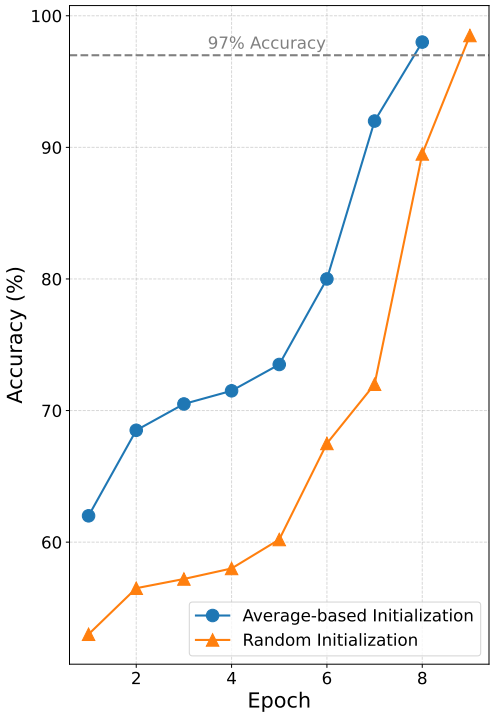


Figure 5: Ablation Study on Embedding Initialization Strategies. We evaluate both initialization strategies on the Textualized Tree classification task using smaller trees (only 5 nodes). **All experiments use our 4-Token Merging Model**. The results show that the model with average-based initialization converges one epoch faster than the model with random initialization when reaching 97% accuracy.

Recursive Encoding: In our setup, we apply the encoder once to compress the input tokens. A natural extension is to apply the encoder recursively – e.g., compress 8 tokens into 4, then compress those 4 into 2 – enabling progressively shorter representations. This hierarchical compression approach may further improve efficiency while preserving performance.

Adaptive Compression: In this paper, we apply uniform compression over every fixed group of K tokens, regardless of the statistical properties of different K-grams. An adaptive strategy – such as compressing 2 tokens in high-frequency regions and 16 in low-frequency ones – could further improve the compression ratio while maintaining or even boosting performance.

6 Conclusions

In this paper, we introduced K-Token Merging, a latent-space compression framework that merges groups of tokens into single embeddings to reduce the effective input length of LLMs. By combining a lightweight encoder with LoRA adaptation, our method enables LLMs to process compressed inputs while preserving standard generation. Experiments across synthetic, natural-language, and code tasks demonstrate up to 75% input length reduction with limited performance degradation. Together, these results suggest that exploiting redundancy in the embedding space is a promising direction for improving the efficiency of long-context LLM inference.

Listing 2: Output code. Our model follows the instructions to add an additional plot of cosine function. It reuse the previously defined variable "x_vals" (highlighted in yellow) as the function domain.

```
import math
import matplotlib.pyplot as plt
import numpy as np

# Define the function
def compute_sin(x):
    return np.sin(x)

# Generate input values
x_vals = np.linspace(-2 * np.pi, 2 * np.
    pi, 500)
y_vals = compute_sin(x_vals)

# Plotting
plt.figure(figsize=(8, 4))
plt.plot(x_vals, y_vals, label='sin(x)')
plt.show()

# Plotting the "cos" function
plt.figure(figsize=(8, 4))
plt.plot(x_vals, np.cos(x_vals), label='
    cos(x)')
plt.legend()
plt.show()
```

Limitations

Our work has the following limitations that we leave for future research:

No compression during generation. Our method compresses only input tokens and does not compress generated tokens, limiting efficiency gains as the output sequence grows.

Fixed compression ratio. We apply uniform compression to every group of K tokens regardless of corpus statistics. Adaptive strategies (e.g., smaller K in dense regions and larger K in sparse ones) could further improve compression while maintaining performance.

Limited model coverage. Most experiments use Qwen 2.5 0.5B, with an additional scaling experiment on Qwen Coder 7B. Evaluation across more model families, parameter scales, and architectures is needed to establish broader generality.

References

Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. 2022. Token merging: Your vit but faster. *arXiv preprint arXiv:2210.09461*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. 2024. xrag: Extreme context compression for retrieval-augmented generation with one token. *Advances in Neural Information Processing Systems*, 37:109487–109516.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. 2023. Adapting language models to compress contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3829–3846.

Frederik Roland Christiansen, Linus Nørgaard Hollensberg, Niko Bach Jensen, Kristian Julsgaard, Kristian Nyborg Jespersen, and Ivan Nikolov. 2024. Exploring presence in interactions with llm-driven npcs: a comparative study of speech recognition and dialogue options. In *Proceedings of the 30th ACM Symposium on Virtual Reality Software and Technology*, pages 1–11.

Yu-Neng Chuang, Tianwei Xing, Chia-Yuan Chang, Zirui Liu, Xun Chen, and Xia Hu. 2024. Learning to compress prompt in natural language formats. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7756–7767.

Samuel Rhys Cox and Wei Tsang Ooi. 2023. Conversational interactions with npcs in llm-driven gaming: Guidelines from a content analysis of player feedback. In *International Workshop on Chatbot Research and Design*, pages 167–184. Springer.

Jun Gao, Qi Lv, Zili Wang, Tianxiang Wu, Ziqiang Cao, and Wenjie Li. 2024. Uniicl: An efficient unified framework unifying compression, selection, and generation. *arXiv preprint arXiv:2405.17062*.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. 2023. In-context autoencoder for context compression in a large language model. *arXiv preprint arXiv:2307.06945*.

John Harvill, Ziwei Fan, Hao Wang, Yizhou Sun, Hao Ding, Luke Huan, and Anoop Deoras. 2025. Lossless token sequence compression via meta-tokens. *arXiv preprint arXiv:2506.00307*.

Yupeng Hou, Jiacheng Li, Zhankui He, An Yan, Xiusi Chen, and Julian McAuley. 2024. Bridging language and items for retrieval and recommendation. *arXiv preprint arXiv:2403.03952*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1658–1677.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72.

Sathvik Joel, Jie Wu, and Fatemeh Fard. 2024. A survey on llm-based code generation for low-resource and domain-specific programming languages. *ACM Transactions on Software Engineering and Methodology*.

Hoyoun Jung and Kyung-Joong Kim. 2024. Discrete prompt compression with reinforcement learning. *IEEE Access*, 12:72578–72587.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pages 3045–3059.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. 2023. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 6342–6353.

Zongqian Li, Yinhong Liu, Yixuan Su, and Nigel Collier. 2025. Prompt compression for large language models: A survey. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7182–7195.

Barys Liskavets, Maxim Ushakov, Shuvendu Roy, Mark Klibanov, Ali Etemad, and Shane K Luke. 2025. Prompt compression with context-aware sentence encoding for fast and improved llm inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24595–24604.

Junyi Liu, Liangzhi Li, Tong Xiang, Bowen Wang, and Yiming Qian. 2023. Tcra-llm: Token compression retrieval augmented large language model for inference cost reduction. In *Findings of the association for computational linguistics: EMNLP 2023*, pages 9796–9810.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Jesse Mu, Xiang Li, and Noah Goodman. 2023. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36:19327–19352.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 workshop on instruction tuning and instruction following*.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, and 1 others. 2024. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. *arXiv preprint arXiv:2403.12968*.

A Yang Qwen, Baosong Yang, B Zhang, B Hui, B Zheng, B Yu, Chengpeng Li, D Liu, F Huang, H Wei, and 1 others. 2024. Qwen2. 5 technical report. *arXiv preprint*.

Matheus Ferracciú Scatolin and Helio Pedrini. 2026. Stellar: A structured, trustworthy, and explainable llm-led architecture for reliable customer support. *Journal of the Brazilian Computer Society*, 32(1):128–144.

Shivam Shandilya, Menglin Xia, Supriyo Ghosh, Huiqiang Jiang, Jue Zhang, Qianhui Wu, Victor Rühle, and Saravan Rajmohan. 2025. Taco-rl: Task aware prompt compression optimization with reinforcement learning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1582–1597.

Farooq Shareef. 2024. Enhancing conversational ai with llms for customer support automation. In *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, pages 239–244. IEEE.

Zirui Tang, Weizheng Wang, Zihang Zhou, Yang Jiao, Bangrui Xu, Boyu Niu, Dayou Zhou, Xuanhe Zhou, Guoliang Li, Yeye He, and 1 others. 2025. Llm/agent-as-data-analyst: A survey. *arXiv preprint arXiv:2509.23988*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Da Yu, Edith Cohen, Badih Ghazi, Yangsibo Huang, Pritish Kamath, Ravi Kumar, Daogao Liu, and Chiyuan Zhang. 2025. Scaling embedding layers in language models. *arXiv preprint arXiv:2502.01637*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, and 1 others. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2):1–124.