CODEPROMPTZIP: Compressing Code Prompt for Retrieval-Augmented Generation in Coding Tasks with LMs

Anonymous ACL submission

Abstract

Retrieval-Augmented Generation (RAG) enhances coding tasks by incorporating retrieved code examples into prompts. However, lengthy prompts—often exceeding tens of thousands of tokens—introduce challenges related to limited context windows of language models (LMs) and high computational costs. Existing prompt compression techniques focus on natural language, lacking tailored solutions for code. To address the gap, we propose CODEPROMPTZIP, a framework that compresses code examples before integrating into RAG workflows. Our framework employs a type-aware, priority-driven strategy to construct training samples for training code compression model. By using program analysis, we identify token types (e.g., Identifier) and perform ablation analysis to rank their removal priorities based on their impact on task performance. We then train a small LM as the compressor on these samples, enabling flexible compression conditioned on specified ratios while minimizing performance degradation. Specially, the compressor's architecture is augmented with a copy mechanism, allowing tokens to be directly copied from the original code snippets. Evaluation results show that CODEPROMPTZIP surpasses SOTA entropy-based and distillation-based baselines, improving by 23.4%, 28.7%, and 8.7% over the best baseline for Assertion Generation, Bugs2Fix, and Code Suggestion, respectively.

1 Introduction

Retrieval-Augmented Generation (RAG) for language models (Lewis et al., 2020; Izacard et al., 2023; Xu et al., 2024) has shown remarkable performance on knowledge-intensive tasks, particularly in coding domains (Nashid et al., 2023; Chen et al., 2024; He et al., 2024), by incorporating retrieved code examples into input prompts. However, such prompts often span tens of thousands of tokens, which creates challenges due to the limited context window of LMs and the high cost of processing long prompts with proprietary services like GPT-4 ($2.50 per million tokens).

Prompt compression offers a promising solution for efficient LM utilization by retaining essential information while reducing prompt length (Chang et al., 2024). Although existing studies have achieved promising results for natural language (NL) tasks, including language modeling (Xu et al., 2024; Chevalier et al., 2023; Mu et al., 2023), question-answering (Jung and Kim, 2024), and summarization (Jiang et al., 2023a; Li, 2023), there is no compressor specifically for coding tasks. To address this gap, we introduce CODEPROMPTZIP, a framework to train a code-specific compressor to compress code examples for RAG-based coding tasks.

We propose using a small LM (i.e., CodeT5 (Wang et al., 2021), 775M) as the compressor to compress code examples. The LM-based compressor captures the probabilistic relationships between code tokens without being constrained by strict syntax, making our framework applicable to incomplete code. The generated compressed examples aim to be lightweight yet effective, ensuring minimal impact on the base LM's ability to produce high-quality outputs. To provide flexibility, the compressor accepts original code examples and desired compression ratios as input, generating examples that align with specified constraints. However, training the compressor introduces two key challenges: ❶ Constructing suitable datasets tailored for code compression. ❷ Designing a compressor architecture that effectively supports code compression while allowing compression ratio control.

To address ❶, we propose a type-aware, priority-driven method to construct code compression datasets. This approach leverages the observation that different token types (e.g., Identifier) in code examples have varying impacts on generation quality. Using program analysis tools (Zhang et al.,

2024), tokens are first categorized by their type. Next, we perform an ablation analysis to measure the impact of each type of tokens and establish a hierarchy of removal priorities based on their impact on performance degradation. Finally, a greedy strategy is employed to iteratively remove higher priority tokens and generate compressed code snippets with varying compression ratios.

To address ❷, we enhance the base CodeT5 architecture with a copy mechanism (See et al., 2017; Zhang et al., 2021), which enables the model to directly copy tokens from the source. Since the compressed code is fully derived from the original, this mechanism introduces a copy distribution over source tokens to guide the token generation from the source sequence during decoding. Additionally, we extend the vocabulary by incorporating special tokens (e.g., <Ratio>), allowing the model to condition on specified compression ratios and adaptively learn compression at varying levels during training.

We evaluated CODEPROMPTZIP by compressing code examples in three RAG-based coding tasks, i.e., Assertion Generation (Nashid et al., 2023), Bugs2Fix (Lu et al., 2021), and Code Suggestion (Chen et al., 2024). CODEPROMPTZIP effectively maintains performance while reducing prompt lengths. CODEPROMPTZIP demonstrates improvements over both SOTA entropy-based (e.g., LLMLingua (Jiang et al., 2023a)) and distillation-based baselines (e.g., RECOMP (Xu et al., 2024)). CODEPROMPTZIP achieves an improvement of 23.4%, 28.7%, and 8.7% over the best baseline for three coding tasks Assertion Generation, Bugs2Fix, and Code Suggestion, respectively.

We make the following contributions.

- We first observe that different types of tokens have varying impacts on final generation quality. Based on this, we propose a novel prompt compression framework designed for compressing code examples.
- We developed a copy-enhanced LM as the compressor to compress code examples effectively and allow compression ratio control.
- Our approach achieves significant performance improvements over SOTA baselines and demonstrates generalization across different language models and tasks.

2 Related Work and Background

2.1 Related work

Prompt compression methods can be broadly classified into two types: **soft prompts** and **discrete prompt compression** (Chang et al., 2024). **Soft prompts** learn embeddings that encode either task instructions (Mu et al., 2023) or example documents (Chevalier et al., 2023). For example, Mu et al. (2023) condense prompt instructions into reusable "gist" vectors, while Chevalier et al. (2023) compress long documents into learnable context vectors. However, soft prompts face limitations in cross-model compatibility and require gradient access to base LMs, making them impractical for API-based proprietary LM services.

Recent research has focused on **discrete prompt compression**, which retains key tokens from the original prompt while eliminating less informative content. This approach enhances compatibility with black-box or proprietary LMs. Notable techniques include **entropy-based** and **knowledge distillation** methods. **Entropy-based** methods, such as LLMlingua and LongLLMlingua (Jiang et al., 2023a,b), use small LMs to estimate the information entropy of tokens, filtering out low-value content. However, they rely on heuristic metrics that may not align well with compression objectives. **Knowledge distillation** leverages large LMs like GPT-4 (Achiam et al., 2023) to generate compressed summaries, which are then used to fine-tune smaller LMs as compressors. For instance, Xu et al. (2024) trained a T5 model on GPT-3.5-turbo summaries, while Pan et al. (2024) employed a transformer encoder to classify tokens for extraction. Despite their effectiveness, distillation methods struggle with maintaining strict compression ratios and entail high costs due to reliance on proprietary LMs.

Although code is a subset of natural language, it exhibits unique features, such as type information (Zhang et al., 2024). Different token types encapsulate distinct symbolic and syntactic information. For example, **Identifier** tokens reflect developers' intent, while **Symbol** tokens define delimiters and operations. A recent work (Yang et al., 2024a) primarily targets the natural language parts (i.e., docstrings) in coding task prompts rather than addressing the compression of code itself. To the best of our knowledge, we are the first to focus on compressing the code.

2.2 Problem Formulation

Referring to Jiang et al., 2023a, we modify and reformulate prompt compression. For a coding task $\mathcal{T}$, given an original prompt, denoted as $\boldsymbol{x} = (\boldsymbol{x}_1^{code}, ..., \boldsymbol{x}_N^{code}, \boldsymbol{x}^{ques})$, where $\boldsymbol{x}_i^{code}$ represents ith

code example [1], N represents number of shots, and $\boldsymbol{x}^{ques}$ represents the question. We aim to compress the code examples to reduce token count while retaining critical information for the question. Formally, the compression is performed by a compressor $\mathcal{LM}_{\mathcal{C}}$, acting as a function:

$$\widetilde{\boldsymbol{x}}_i^{code} = \mathcal{LM}_{\mathcal{C}}(\boldsymbol{x}_i^{code}, \tau_{code}, \mathcal{T}) \tag{1}$$

where $\tau_{code} = 1 - |\widetilde{\boldsymbol{x}}_i^{code}|/|\boldsymbol{x}_i^{code}|$ is the compression ratio for a code snippet. With compressed code examples, the overall prompt is shortened as:

$$\begin{aligned}\widetilde{\boldsymbol{x}} &= \textsc{CodePromptZip}(\boldsymbol{x}) \\ &= (\{\mathcal{LM}_{\mathcal{C}}(\boldsymbol{x}_i^{code}, \tau_{code})\}_{i=1}^{N}, \boldsymbol{x}^{ques})\end{aligned} \tag{2}$$

where the overall ratio is given by $\tau = 1 - \widetilde{\boldsymbol{x}}/\boldsymbol{x}$. The generation of the base language model $\mathcal{BLM}$ with the compressed prompt $\widetilde{\boldsymbol{x}}$ is expected to closely approximate the generation with the original prompt $\boldsymbol{x}$. This can be formulated as:

$$\min_{\widetilde{\boldsymbol{x}}, \tau} \mathrm{KL}\left(P\left(\mathcal{BLM}(\widetilde{\boldsymbol{x}}) \mid \widetilde{\boldsymbol{x}}\right), P\left(\mathcal{BLM}(\boldsymbol{x}) \mid \boldsymbol{x}\right)\right) \tag{3}$$

3 Type-aware Priority Ranking

Before introducing our framework, we outline its motivation. Different tokens in a prompt contribute unevenly to the final output, with less impactful tokens prioritized for removal (Yang et al., 2024b; Li, 2023; Jiang et al., 2023a,b; Xu et al., 2024). In coding tasks, prompts often include code snippets as RAG demonstrations. A key question arises: *Do different types of tokens in code contribute differently to the final results?* If so, this insight can guide code prompt compression. To explore this, we use program analysis (PA) tools to categorize tokens by type and conduct ablation analysis to identify those with minimal impact, guiding efficient prompt compression.

3.1 Type Ablation Analysis

We categorize tokens into five types, based on the taxonomy proposed by Wang et al., 2024: **Symbol**, **Signature**, **Invocation**, **Identifier**, and **Structure** (see Appendix B for detailed descriptions).

We constructed Abstract Syntax Trees (ASTs) using JavaParser (Anonymous, 2019) to identify token types. Tokens of specific types were removed from the retrieved code examples, followed by performing RAG with the type-ablated examples to measure their impact on $\mathcal{BLM}$s performance in downstream tasks. The removal priority of a token type T is defined as follows:

$$Priority(T) = \frac{\tau_{code/T}}{d_T} \tag{4}$$

where $\tau_{code/T}(\%)$ represents the code compression ratio achieved by discarding tokens of type T, d_T (%) denotes the percentage of performance degradation in the evaluation metric caused by the removal of tokens of type T.

Token types that yield a higher ratio of $\tau_{code/t}$ and result in minimal degradation d_t are assigned higher priority for removal. This approach ensures the compression process effectively reduces token length while preserving the generation quality.

3.2 Setup of Downstream Coding Tasks with RAG

To comprehensively assess the impact of different types of tokens, we evaluated them across three datasets and two frozen-parameter $\mathcal{BLM}$s.

Dataset and metric: (i) **Assertion Generation**: The input is a focal method (the method under test) and its partial unit test, while the outputs are assertion statements verifying its correctness. The evaluation metric is the Exact Match Rate, following prior work (Nashid et al., 2023). (ii) **Bugs2Fix**: The input is a buggy method, and the output is a refined version of the method with the bugs fixed. This task is evaluated using CodeBleu (Ren et al., 2020), consistent with the CodexGLUE benchmark (Lu et al., 2021). (iii) **Code Suggestion**: The input consists of a method header (a summary of a function), and the output is a suggested code snippet for the developer based on the header. The task is also evaluated using CodeBleu defined in the original paper (Chen et al., 2024). Table 1 presents the statistics of datasets.

Task	Knowledge Base (Parsable)	Test	Val
Assertion Generation	144,112 (70,433)	18,027	18,816
Bugs2Fix	52,364 (48,903)	6,545	6,546
Code Suggestion	128,724 (89,014)	10,727	5,149

Table 1: Dataset statistics of different coding tasks.

In all tasks, we utilize the code RAG prompt template (Chen et al., 2024; He et al., 2024; Nashid et al., 2023) (see Figure A), and craft task-specific instructions in a one-shot setting. As listed in Table

[1] As improving the retriever is not the focus of this work, we retrieve examples using the SOTA BM25 (He et al., 2024).

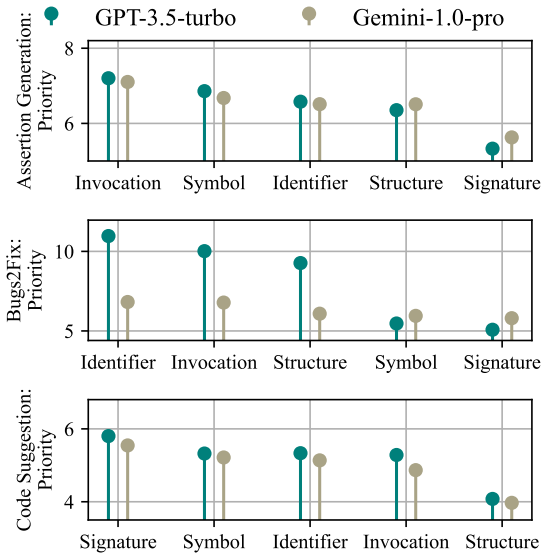


Figure 1: Removal priority of code token types: e.g., Invocation > Symbol in Assertion Generation, and vice versa in Code Suggestion. Priorities are task-specific yet model-agnostic, applicable to both LMs.

1, we follow the original split of the dataset into Train, Validation, and Test partitions. The training partition functions as our knowledge base for example retrieval. Note that some code examples that yield parsing errors in JavaParser due to code incompleteness are classified as **Unparsable**. Due to computational resource constraints, we randomly sample 2,000 instances from both the validation and test sets for our experiments. The sampled validation set is used to study example removal priority, while the sampled test set serves to evaluate performance.

.

Base LMs: The in-context learning capabilities of large language models enable them to utilize query-related documents to produce outputs that better align with the instructions. To investigate whether the impact of different token types is consistent across models, we tested the constructed prompts on two large-scale $\mathcal{BLM}s$: GPT-3.5-turbo and Gemini-1.0-Pro. We set temperature to 0 to ensure enhanced stability across experiments.

3.3 Observation

Figure 1 presents ablation analysis results using a log y-axis to normalize priority scores. The plots reveal type hierarchies of tokens in removal priority, arranged in descending order. This visualization highlights that higher-priority token types should be preferentially removed, providing an intuitive representation of the token-type removal strategy. In addition, the hierarchies are consistent across $\mathcal{BLM}s$ but exhibit in-task variations, suggesting the cross-model adaptability of priority-driven code compression.

4 Methodology

As illustrated in Figure 2, CODEPROMPTZIP operates in two phases. In the training phase, we first derive a type-aware priority ranking for a specific task $\mathcal{T}$ (Sec. 3). Using this ranking, we implement a priority-driven strategy (Algorithm 1): *tokens in higher-priority types are discarded before those in lower-priority types*. This process transforms $(\boldsymbol{x}i^{code}, \tau_{code}, \mathcal{T})$ into $\tilde{\boldsymbol{x}}_i^{code}$. We then train $\mathcal{LM}_\mathcal{C}$ on the constructed dataset to learn the sequence-to-sequence compression task.

The design of the learning-based $\mathcal{LM}_\mathcal{C}$ enhances **applicability**. While Algorithm 1 can directly output compressed code examples, its implementation relies on JavaParser for token labeling and removal, restricting its use to unparsable code. However, as shown in Table 1, unparsable code examples are common in coding tasks.

The $\mathcal{LM}_\mathcal{C}$ processes code sequences as probabilistic relations (Xu et al., 2024; Pan et al., 2024) rather than relying strictly on exact syntax, enables our framework to handle both parsable and unparsable code examples while tolerating compile and parse errors (Yadavally et al., 2024).

In the inference phase, given a query, the $\mathcal{LM}_\mathcal{C}$ accepts a specified τ_{code} and the original retrieved code example to generate compressed code that retains the most critical tokens. These compressed examples are then aggregated into a prompt and passed to the $\mathcal{BLM}$ to generate the final output.

4.1 Code Compression Dataset Construction

The workflow of Algorithm 1 is as follows. Line 1 initializes a priority queue to store tokens alongside their priorities. Lines 2–5 assign priorities to tokens based on their type, with tokens in the same type ranked by Term Frequency (TF) within the example. Frequently occurring tokens are prioritized for removal, as they are more likely to be redundant. Tokens belonging to multiple types are assigned to the category with the lowest removal priority, while out-of-type tokens are removed last, preserving potentially critical tokens. Line 6 initializes an empty set, removedTokens, to track removed tokens. Line 7 calculates the number of tokens to remove as a fraction of the total, determined by the

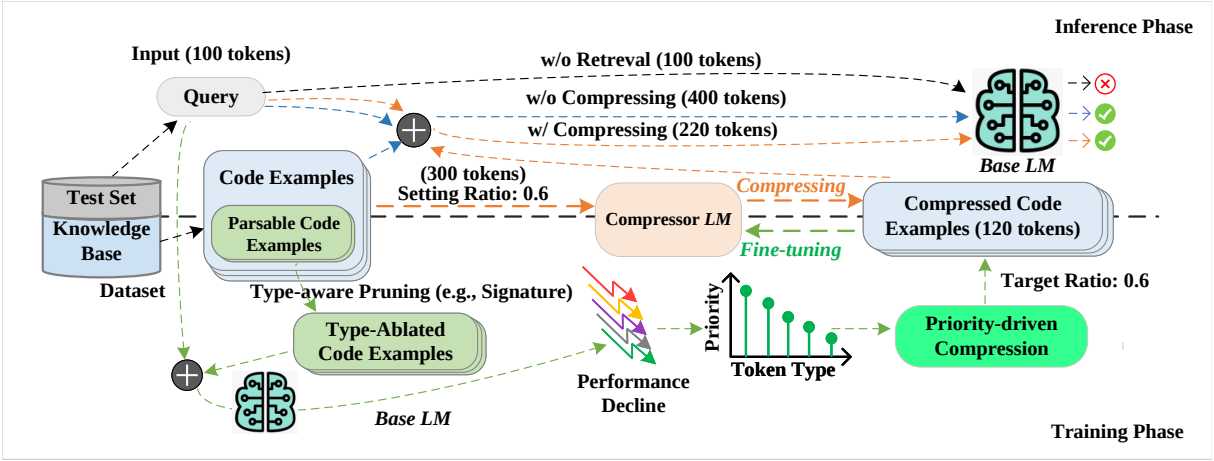


Figure 2: Framework of CODEPROMPTZIP.

Algorithm 1: Priority-driven Greedy Algorithm for Dataset Construction

Input: $\boldsymbol{x}_i^{code} = \{x_j\}_{j=1}^{L}$, τ_{code}, type priorities of $\mathcal{T}$.
Output: $\widetilde{\boldsymbol{x}}_i^{code}$.

1: Initialize a priority queue *pq*.
2: **for** each token $x_j \in \boldsymbol{x}_i^{code}$ **do**
3: Assign priority to x_j (*Prioritize the drop of high-frequency tokens in prioritized type*).
4: Insert x_j into *pq*.
5: **end for**
6: *removedTokens* $\leftarrow \emptyset$.
7: $L_{rm} \leftarrow \lfloor \tau_{code} \cdot L \rfloor$.
8: $\widetilde{L}_{rm} \leftarrow 0$.
9: **while** $\widetilde{L}_{rm} < L_{rm}$ **do**
10: $x_j \leftarrow$ *pq.pop()*.
11: *removedTokens* $\leftarrow$ *removedTokens* $\cup \{x_j\}$.
12: $\widetilde{L}_{rm} \leftarrow \widetilde{L}_{rm} + 1$.
13: **end while**
14: $\widetilde{\boldsymbol{x}}_i^{code} \leftarrow \boldsymbol{x}_i^{code} \setminus$ *removedTokens*.
15: **return** $\widetilde{\boldsymbol{x}}_i^{code}$.

specified τ_{code}. Lines 9–13 iteratively remove the highest-priority tokens from the queue until the required number is removed. Line 14 constructs the modified training sample by excluding tokens in removedTokens from the original sequence. This iterative, priority-driven approach ensures the compressed code retains essential tokens while meeting the specified compression ratio.

Using Algorithm 1, we constructed a code compression dataset for training compressors (see dataset statistics in Appendix C).

4.2 Compressor Architecture

With the code compression dataset, we fine-tune an encoder-decoder model, $\mathcal{LM}_{\mathcal{C}}$, to effectively compress code examples. We adopt CodeT5 (Wang et al., 2021) as our base model and introduce two key modifications to its architecture. First, we extend the input vocabulary with task-indicative tokens such as <*ASSERTION*>, <*BUGS2FIX*>, and <*SUGGESTION*>, which are added at the beginning of the input sequence to explicitly indicate the task context. This design allows our model to be extended to more coding tasks. Additionally, to enable $\mathcal{LM}_{\mathcal{C}}$ to condition on flexible τ_{code} settings, we introduce special tokens <*Ratio*>, </*Ratio*>, <*Compress*>, and </*Compress*>. These tokens signal the model to generate compressed code snippets tailored to the specified τ_{code} and task. Moreover, we incorporate a copy mechanism (See et al., 2017; Zhang et al., 2021) into the architecture, allowing the model to directly copy tokens from the input sequence. This modification aligns with the extractive nature of the code compression task, where the outputs are derived entirely from the inputs.

The detailed architecture is shown in Figure 3. This mechanism is implemented using a copy module that computes the probability of copying each generated token directly from the input, rather than generating it from entire vocabulary. At first, the tokens of the original code sequence $\boldsymbol{x}_i^{code} = \{x_j\}_{j=1}^{|\boldsymbol{x}_i^{code}|}$ are fed into the encoder, producing a sequence of encoder hidden states $\mathbf{h} = \{h_j\}$. In the decoder, the last cross-attention matrix $\mathbf{A} \in \mathbb{R}^{l_{\text{tgt}} \times l_{\text{src}}}$ represents the attention distribution over the source sequence during decoding. Each row $\mathbf{a}^t \in \mathbb{R}^{l_{\text{src}}}$ corresponds to the attention weights assigned to the source sequence at decoding step t. Here, l_{src} and l_{tgt} denote the maximum input and output lengths, respectively. The attention distribution not only guides the decoder's focus for each source token, but also allows tokens to be copied from the source sequence by sampling from the attention distribution. Next, the

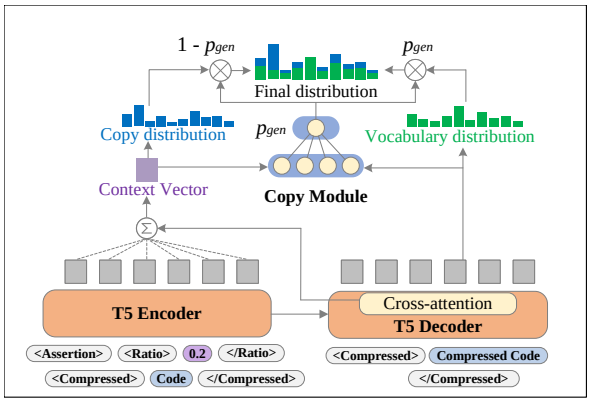


Figure 3: Illustration of copy mechanism on CodeT5.

attention distribution generates a weighted sum of the encoder hidden states, known as context vectors $\mathbf{h}^*$:

$$\mathbf{h}_t^* = \sum_i a_i^t \mathbf{h}_i, \tag{5}$$

The context vector $\mathbf{h}_t^*$ represents a fixed size summary of what has been read from the source. Then the context vector is concatenated with the decoder state $\mathbf{s}_t$, and passed through a copy module to calculate the generation likelihood $p_{gen} \in [0, 1]$ at this step:

$$p_{gen} = \sigma(\mathbf{W}_{gen} \cdot [\mathbf{h}_t^*, \mathbf{s}_t] + \mathbf{b}_{gen}) \tag{6}$$

where $\mathbf{W}_{gen}$ and $\mathbf{b}_{gen}$ are learnable parameters of the linear copy module. Here, p_{gen} corresponds to the probability of generating tokens from the vocabulary, while $(1-p_{gen})$ denotes the probability of copying tokens from the input.

Next, we calculate the copy distribution by summing the attention weights a_i^t for all positions i where the input token x_i match the target token y:

$$P_{copy}(y) = \sum_{i:x_i=x} a_i^t \tag{7}$$

The generation probability is computed through the language model head connected to the decoder's output, defined as:

$$P_{vocab}(y) = Softmax(\mathbf{W}_{head} \cdot \mathbf{s}_t + \mathbf{b}_{head}) \tag{8}$$

where $\mathbf{W}_{head}$ and $\mathbf{b}_{head}$ denote the weight matrix and bias vector of the head network, respectively.

Finally, the output distribution is computed by interpolating between generation distribution P_{vocab} and copy distribution P_{copy}:

$$P(y) = p_{gen} P_{vocab}(y) + (1 - p_{gen}) P_{copy}(y) \tag{9}$$

During training, we use the Cross-Entropy Loss to maximize the likelihood of the target sequence. The loss function is defined as:

$$\mathcal{L} = -\sum_{t=1}^{T} y_t \log(\hat{y}_t) \tag{10}$$

where y_t is the ground-truth token at step t, and $\hat{y}_t$ is the predicted probability of that token.

We train the model by using the AdamW optimizer with a batch size of 16, a learning rate of 5e-5, and 1,000 warmup steps for 10 epochs.

5 Experimental Setting

5.1 Research Questions

- RQ1: How effective is CODEPROMPTZIP compared to NL-specific prompt compression methods on coding tasks?
- RQ2: What is the trade-off between compression ratio and number of shots in CODEPROMPTZIP's performance?
- RQ3: How effective is CODEPROMPTZIP in controlling compression ratios?
- RQ4: How does CODEPROMPTZIP perform across different $\mathcal{BLM}$s?
- RQ5: How does CODEPROMPTZIP perform on unparsable code snippets?

In RQ1, we compare generation quality by the $\mathcal{BLM}$ using compressed prompts from existing approaches. RQ2 examines the impact of two key hyper-parameters, τ_{code} and number of shots, analyzing the trade-off between using highly compressed examples versus fewer complete ones within a fixed budget. RQ3 examines the ability of CODEPROMPTZIP on controlling compression ratios. RQ4 evaluates CODEPROMPTZIP's performance across different $\mathcal{BLM}$ to test its generalization. RQ5 explores scenarios with unparsable code, demonstrating the robustness of CODEPROMPTZIP as a learning-based framework.

5.2 Baselines and Oracle

We compare our approach against four state-of-the-art prompt compression baselines: LLMLingua (Jiang et al., 2023a), LongLLMLingua (Jiang et al., 2023b), LLMLingua-2 (Pan et al., 2024), and RECOMP (Xu et al., 2024), with detailed descriptions provided in Sec. 2.1. For reference, we also evaluate prompts without retrieval or compression. Additionally, we include Oracle, where we iteratively remove tokens directly based on their type-aware priority ranking, without using compressor model

(as the approach used to construct training dataset in Section 4.1), as a program analysis-based baseline for code examples.

5.3 Datasets and Metrics

We evaluated the performance of CODEPROMPTZIP on the same three coding tasks, using the same prompt template and metrics, as presented in Sec. 3.2.

5.4 Base LMs

Concrete compression offers the advantage of transferability across various $\mathcal{BLM}$s (Xu et al., 2024; Jung and Kim, 2024). For RQ1, RQ2, RQ3, and RQ5, we conducted experiments on GPT-3.5-turbo. In RQ4, to evaluate the generalization of CODEPROMPTZIP, we conducted experiments on two additional $\mathcal{BLM}$s: the open-source CodeLlama-13B (Roziere et al., 2023) and the proprietary LM service Gemini-1.0-pro (Team et al., 2023).

6 Results

6.1 RQ1: Comparisons with Baselines

Table 2 summarizes the results of different approaches for compressing retrieval-augmented prompts across three coding tasks, evaluating metrics such as token count, τ (overall compression ratio), and task performance, with the best results reported. The baseline approach without retrieved code examples (w/o retrieval) performs significantly worse than approaches with retrieval, highlighting the importance of RAG in enhancing $\mathcal{BLM}$ performance on coding tasks. CODEPROMPTZIP demonstrates improvements over both entropy-based and distillation-based baselines, improving by 23.4%, 28.7%, and 8.7% over the best baseline for Assertion Generation, Bugs2Fix, and Code Suggestion, respectively.

Comparison with Oracle highlights the learning outcomes derived from the code compression dataset. CODEPROMPTZIP closely approaches Oracle-level performance without requiring JavaParser tools or parsable code snippets, falling only 4.1% short in Exact Match for Assertion Generation and 4.9% in CodeBleu for Bugs2Fix, while achieving nearly identical performance in Code Suggestion.

The ablation study shows the consistent contributions of the copy mechanism to the enhancement of CodeT5, with a 1.2% Exact Match increase for Assertion Generation, CodeBleu improvements of

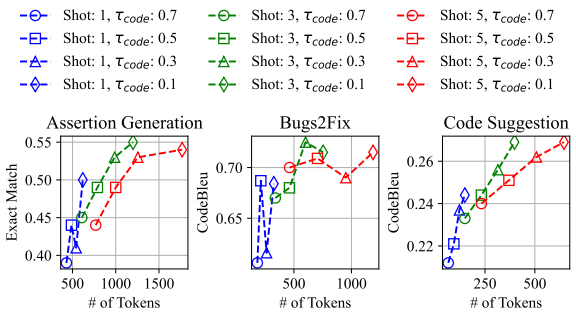


Figure 4: Trade-off between keeping more tokens in a single example or including more examples.

5.2% for Bugs2Fix, and 3.2% for Code Suggestion. While uncompressed prompts achieve the highest quality metrics, they incur a significant token cost.

6.2 RQ2: Trade-off between τ_{code} and Shots

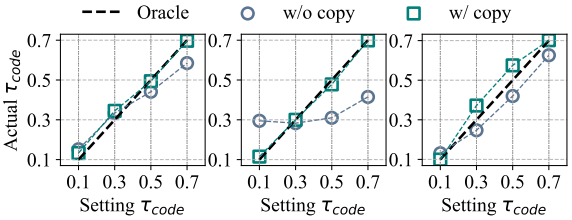
The objective of prompt compression is to minimize the number of tokens fed to the $\mathcal{BLM}$, while preserving acceptable generation quality. Given a fixed token budget, a trade-off arises between including fewer, less-compressed examples and more highly-compressed ones. Figure 4 illustrates this balance. **In general, appending fewer examples, with each example allocated more tokens, achieves better performance than increasing the number of shots while allocating fewer tokens per shot.** For instance, in Assertion Generation, with a token budget of 500, a single example compressed at $\tau_{code} = 0.1$ outperforms three examples compressed at $\tau_{code} = 0.7$. Additionally, to achieve a fixed performance level, choosing fewer shots with a lower τ_{code} is more cost-effective, balancing token efficiency and performance.

6.3 RQ3: Compression Ratio Control

Our LM-based compressors utilize an extended vocabulary and accept τ_{code} as input, enabling adaptive compression of code examples to meet the desired ratio. Figure 3 illustrates the relation between the specified τ_{code} and the actual achieved values. The dotted line (Oracle) represents the standard outcome, and CODEPROMPTZIP closely aligns with this benchmark. In contrast, compressors based on the original CodeT5 architecture (w/o the copy) struggle to produce outputs that match the desired ratio. Table 6 also provides specific results under varying τ_{code} configurations, further demonstrating the effectiveness of CODEPROMPTZIP and the critical role of the copy mechanism in achieving accurate compression ratio control.

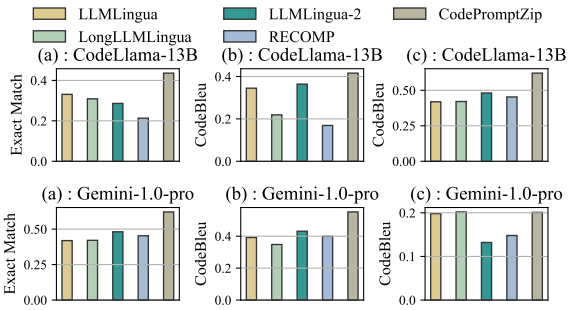
Table 2: Results on three coding tasks using GPT-3.5-turbo as the $\mathcal{BLM}$. To ensure fair comparison with baselines that lack a specified compression rate, we set CODEPROMPTZIP's compression rate to 0.3, keeping it similar to or higher than the baselines. Note that higher metric values indicate better performance, while a higher τ (%) reflects a greater proportion of tokens removed from the prompt.

	Assertion Generation			Bugs2Fix			Code Suggestion		
Approach	**# tokens**	τ(%)	**Exact Match(%)**	**# tokens**	τ(%)	**CodeBleu(%)**	**# tokens**	τ(%)	**CodeBleu(%)**
w/o retrieval	334	46.6	23.9	122	66.3	41.7	29	82.6	14.2
Entropy-based									
LLMLingua	482	22.9	33.8	286	20.9	41.9	125	25.1	21.8
LongLLMLingua	474	24.2	34.1	287	20.6	42.1	126	24.1	21.2
Knowledge Distillation									
LLMLingua-2	469	25.1	21.2	282	21.9	48.1	134	19.3	21.7
RECOMP	465	25.6	23.4	268	25.9	45.3	132	20.9	21.0
Ours, Setting τ_{code}-0.3, 1-shot									
CODEPROMPTZIP w/o Copy	447	28.5	40.9	267	26.2	56.7	131	21.7	20.5
CODEPROMPTZIP	440	29.7	42.1	262	27.4	61.9	121	27.5	23.7
Oracle	454	27.4	46.2	276	23.5	66.8	120	28.1	23.8
w/o Compression	626	0.0	50.5	362	0.0	81.4	167	0.0	24.7



(a) Assertion Generation (b) Bugs2Fix (c) Code Suggestion

Figure 5: Compression ratio control.



(a) Assertion Generation (b) Bugs2Fix (c) Code Suggestion

Figure 6: Performance of the proposed CODEPROMPTZIP across different $\mathcal{BLM}$s.

6.4 RQ4: Transferability with Different $\mathcal{BLM}$

CODEPROMPTZIP consistently outperforms baselines across studied base LMs CodeLlama-13B and Gemini-1.0. Figure 6 compares the performance among the studied prompt compression techniques across two additional base LMs and all three tasks. In comparison, baseline methods exhibit varying effectiveness, occasionally suffering significant performance drops (e.g., RECOMP on CodeLlama for Bug2fix). The consistent superiority underscores CODEPROMPTZIP 's robustness and effectiveness as a transferable, generalized compression method for code-related tasks.

Table 3: Results on unparsable code examples.

	Assertion Generation		Bugs2Fix		Code Suggestion	
Approach	τ(%)	**Exact Match(%)**	τ(%)	**CodeBleu(%)**	τ(%)	**CodeBleu(%)**
Omit 1% at end, 1-shot Setting code-0.1						
CODEPROMPTZIP w/o Copy	29.1	39.7	26.4	55.4	21.9	19.4
CODEPROMPTZIP	30.1	42.0	27.6	61.9	28.2	23.9
Oracle	N/A	N/A	N/A	N/A	N/A	N/A
Omit 3% at end, 1-shot Setting code-0.3						
CODEPROMPTZIP w/o Copy	29.5	38.4	26.5	50.8	22.0	18.7
CODEPROMPTZIP	31.2	0.417	28.0	61.0	29.1	22.6
Oracle	N/A	N/A	N/A	N/A	N/A	N/A

6.5 RQ5: Applicability on Unparsable Code

Table 3 presents the results of our experiments. Drawing inspiration from Yadavally et al., 2024, we removed a specified percentage of tokens from the end of code examples to render them unparsable and evaluated the effectiveness of our framework. For this experiment, we remove 1% and 3% of tokens from the end. When setting $\tau_{code} = 0.3$, the exact match rate showed only a slight decrease, from 42.1% (as shown in Table 2) for parsable code to 42.0% and from 42.1% to 41.7% when removing 1% and 3% of tokens, respectively. In contrast, the Oracle method, which depends on complete code and ASTs, is not applicable (N/A). The results demonstrate the capability of our compressor for real-world scenarios where code completeness cannot always be ensured.

7 Conclusion

This paper presents CODEPROMPTZIP, a framework designed to compress retrieved code examples before incorporating them into prompts. The proposed compressor leverage copy-enhanced LMs and are trained on dedicated datasets. Experimental results demonstrate that CODEPROMPTZIP significantly improves the efficiency of retrieval-augmented LMs while maintaining minimal performance degradation. Note that our framework is not limited to RAG, it could be applied to any prompt that contains code examples.

8 Limitations

Need for Extra Training for New Coding Tasks: This study focuses on learning code compression for three method-level coding tasks. Similar to other task-aware compressors (e.g., Jiang et al., 2023b), the removal priorities in our approach depend on the downstream coding task. If CODEPROMPTZIP is to be applied to other coding tasks, such as repository-level tasks (Ding et al., 2024), where token priorities differ significantly, additional training of the compressor is required. However, as shown in Figure 1, while priority rankings are task-specific, certain patterns emerge consistently. For example, **Identifier** tokens exhibit a higher removal priority than **Structure** tokens across all tasks. We show the out-of-domain capability of our compressor by cross-task experiment in Appendix D.

Generalizability of Our Findings: This study focuses exclusively on Java and method-level tasks. Since programming languages like Python also have program analysis tools, CODEPROMPTZIP is applicable to them as well. Future research could extend this work to other tasks and languages. Our experiments utilized GPT-3.5, Gemini, and CodeLlama-13b. We encourage further studies to explore additional base LMs and a broader range of programming languages and coding tasks.

9 Ethical Considerations

The implementation of this work is conducted with transparency, providing full disclosure of all technical details, limitations, and potential issues to the relevant stakeholders. The work avoids any false or misleading claims and ensures no data is fabricated or falsified.

In the interest of public benefit, the authors support reasonable and ethical uses of their intellectual contributions. Both the source code and data are released as free and open-source software and are made available in the public domain.

References

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

Anonymous. 2019. Javaparser. https://github.com/javaparser/javaparser.

anonymous. 2025. Codepromptzip. https://anonymous.4open.science/r/CodePromptZip-6B2B.

Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Tong Xiao, and Jingbo Zhu. 2024. Efficient prompting methods for large language models: A survey. arXiv preprint arXiv:2404.01077.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code search is all you need? improving code suggestions with code search. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. 2023. Adapting language models to compress contexts. Preprint, arXiv:2305.14788.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. Advances in Neural Information Processing Systems, 36.

Pengfei He, Shaowei Wang, Shaiful Chowdhury, and Tse-Hsun Chen. 2024. Exploring demonstration retrievers in rag for coding tasks: Yeas and nays! Preprint, arXiv:2410.09662.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2023. Atlas: Few-shot learning with retrieval augmented language models. Journal of Machine Learning Research, 24(251):1–43.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023a. LLMLingua: Compressing prompts for accelerated inference of large language models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 13358–13376, Singapore. Association for Computational Linguistics.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023b. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. arXiv preprint arXiv:2310.06839.

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

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems, 33:9459–9474.

Yucheng Li. 2023. Unlocking context constraints of llms: Enhancing context efficiency of llms with self-information-based content filtering. arXiv preprint arXiv:2304.12102.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664.

Jesse Mu, Xiang Li, and Noah Goodman. 2023. Learning to compress prompts with gist tokens. In Advances in Neural Information Processing Systems, volume 36, pages 19327–19352. Curran Associates, Inc.

Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pages 2450–2462. IEEE.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. 2024. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In Findings of the Association for Computational Linguistics ACL 2024, pages 963–981, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. arXiv preprint arXiv:2312.11805.

Yan Wang, Xiaoning Li, Tien N Nguyen, Shaohua Wang, Chao Ni, and Ling Ding. 2024. Natural is the best: Model-agnostic code simplification for pre-trained large language models. Proceedings of the ACM on Software Engineering, 1(FSE):586–608.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 8696–8708.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. 2024. RECOMP: Improving retrieval-augmented LMs with context compression and selective augmentation. In The Twelfth International Conference on Learning Representations.

Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024. A learning-based approach to static program slicing. Proc. ACM Program. Lang., 8(OOPSLA1).

Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. 2024a. Less is more: Docstring compression in code generation. arXiv preprint arXiv:2410.22793.

Guang Yang, Yu Zhou, Wei Cheng, Xiangyu Zhang, Xiang Chen, Terry Yue Zhuo, Ke Liu, Xin Zhou, David Lo, and Taolue Chen. 2024b. Less is more: Docstring compression in code generation. Preprint, arXiv:2410.22793.

Tong Zhang, Long Zhang, Wei Ye, Bo Li, Jinan Sun, Xiaoyu Zhu, Wen Zhao, and Shikun Zhang. 2021. Point, disambiguate and copy: Incorporating bilingual dictionaries for neural machine translation. 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), pages 3970–3979.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the perspectives of nlp and software engineering: A survey on language models for code. Preprint, arXiv:2311.07989.

Demonstrations:	**Demonstrations:**	**Demonstrations:**
[START] ### FOCAL_METHOD: {*method under test*} ### UNIT_TEST : {*test method*} ### Assertion : {*assertion statement*} … [END] **Query** [START] ### FOCAL_METHOD : {*tested method*} ### UNIT_TEST : {*test method*} ### Assertion :	[START] ### BUGGY_CODE: {*buggy method*} ### FIXED_CODE: {*repaired method*} … [END] **Query** [START] ### BUGGY_CODE: {*buggy method*} ### FIXED_CODE :	[START] ### METHOD_HEADER: {*header*} ### WHOLE_METHOD: {*body*} … [END] **Query** [START] ### METHOD_HEADER: {*header*} ### WHOLE_METHOD:

(a) Assertion Generation (b) Bugs2Fix (c) Code Suggestion

Figure 7: The illustration of different RAG coding tasks alongside their respective prompt templates.

A Data Availability

We have made our replication package available, which contains all the code and datasets available here (anonymous, 2025).

B Token Taxonomy of Code

(i) **Symbol**: Tokens representing operators, delimiters, and other symbolic elements that define the structure of the code (e.g., `=`, `{`, `;`,).

(ii) **Signature**: Tokens defining the declaration and parameters of methods, critical for understanding the interface and functionality of code components (e.g., `calculate(int x)`).

(iii) **Invocation**: Tokens related to function or method calls, capturing interactions and dependencies within the code.

(iv) **Identifier**: Tokens that serve as variable names, class names, or other user-defined labels, are essential for understanding program semantics.

(v) **Structure**: Tokens associated to loops, conditional, and other flow-control statements, which dictate the logical behavior of the program (e.g., `if`, `for`, `class`).

C Statistics of the Code Compression Dataset for Compressor Training

The constructed dataset includes original code examples paired with compressed code examples, with τ_{code} ranging from 0.1 to 0.9. As shown in Table 4, the total number of training samples is nine times the number of parsable code examples in the knowledge base, reflecting the nine distinct τ_{code} values. The dataset is split into training, validation, and test sets in an 8:1:1 ratio.

Table 4: Statistics of the Code Compression Dataset for Compressor Training.

Task	Total Samples	Split(Training/Test/Validation)
Assertion	70433*9	80%/10%/10%
Bugs2Fix	48903*9	80%/10%/10%
Suggestion	89014*9	80%/10%/10%

D More Results of the Out-of-Domain Capabilities of CODEPROMPTZIP

To evaluate the out-of-domain effectiveness of our compressors, we performed cross-task experiments using $\mathcal{LM}_{\mathcal{C}}$ trained on individual downstream tasks and tested on the other two out-of-domain tasks. Notably, task-specific special tokens (e.g., <ASSERTION>) were not used in these experiments. Table 5 summarizes the results of these cross-task evaluations.

For example, in the first row, compressors trained on the Assertion Generation task are applied to compress code examples from Bugs2Fix and Code Suggestion. The Assertion Generation in-task compressor achieves a CodeBleu score of 50.3% with a τ_{code} of 33.2% on Bugs2Fix, compared to its in-task performance of 61.9% CodeBleu and a τ_{code} of 30.0% . While the compressor demonstrates slightly less precision in achieving the desired compression ratio and exhibits degradation in performance, it remains competitive against other baselines, such as the best-performing LLMlingua2 with a CodeBleu score of 48.1%.

E More Results of Impact of the τ_{code} on the Effectiveness of CODEPROMPTZIP

In Table 6, we present results with varying τ_{code} settings. For compression ratio control, our framework enables the configuration of τ_{code} as an input to $\mathcal{LM}_{\mathcal{C}}$, allowing it to adaptively compress code examples to match the specified ratio and thereby control the overall τ of the prompt. However, when using the same configuration and setup with the original CodeT5 structure (w/o copy), the framework struggles to effectively learn the token removal priority. Consequently, the output occasionally deviates from the specified ratio settings, as observed in the Bugs2Fix 1-shot experiment with $\tau_{code} = 0.1$. This underscores the critical role of the copy mechanism in ensuring compliance with ratio settings.

Regarding the quality of generation in the end tasks, the performance varies with different τ_{code} when a single retrieved example is included. A

Table 5: Cross-task Results: **Bold** font indicates the in-task scenario.

Compressor			(a): Assertion Generation		(b): Bugs2Fix		(c): Code Suggestion	
(a)	(b)	(c)	$\tau_{code}(\%)$	**Exact Match(%)**	$\tau_{code}(\%)$	**CodeBleu(%)**	$\tau_{code}(\%)$	**CodeBleu(%)**
✓			**31.5**	**42.1**	33.2	50.3	19.8	13.4
	✓		28.4	41.9	**30.0**	**61.9**	25.1	15.9
		✓	34.2	34.1	39.9	43.6	**32.2**	**23.7**

Table 6: Results with varying numbers of code snippets and τ_{code} settings of the compressor on the studied tasks. The overall compression ratio τ is achieved by compressing code snippets with τ_{code}.

Approach	Assertion Generation			Bugs2Fix			Code Suggestion		
	$\tau_{code}(\%)$	$\tau(\%)$	**Exact Match(%)**	$\tau_{code}(\%)$	$\tau(\%)$	**CodeBleu(%)**	$\tau_{code}(\%)$	$\tau(\%)$	**CodeBleu(%)**
1-shot, Setting τ_{code}-0.1									
CODEPROMPTZIP w/o Copy	14.2	6.2	48.3	29.5	25.9	59.0	8.1	5.2	23.4
CODEPROMPTZIP	13.3	7.2	50.1	11.5	9.1	68.4	9.9	8.9	24.4
Oracle	10.0	8.8	49.8	10.0	8.1	78.5	10.0	8.4	24.5
1-shot, Setting τ_{code}-0.3									
CODEPROMPTZIP w/o Copy	33.4	28.5	40.9	28.3	26.2	56.7	24.7	21.7	20.5
CODEPROMPTZIP	31.5	29.7	42.1	30.0	27.4	61.9	32.2	27.5	23.7
Oracle	30.0	27.4	46.2	30.0	20.2	66.8	30.0	26.1	23.8
1-shot, Setting τ_{code}-0.5									
CODEPROMPTZIP w/o Copy	44.1	38.3	40.3	31.0	21.1	56.5	42.9	34.4	23.5
CODEPROMPTZIP	49.4	44.9	45.1	47.8	41.4	68.7	57.5	42.9	22.1
Oracle	50.0	45.2	42.1	50.0	43.9	67.1	50.0	41.9	23.1
1-shot									
w/o compression	0.0	0.0	50.5	0.0	0.0	81.4	0.0	0.0	24.7
3-shot, Setting τ_{code}-0.5									
CODEPROMPTZIP w/o Copy	44.2	39.4	47.2	33.2	24.5	67.2	41.2	39.1	23.5
CODEPROMPTZIP	49.3	43.8	48.9	50.9	42.3	68.0	57.3	42.3	24.4
Oracle	50.0	41.1	52.6	50.0	41.9	68.9	50.0	42.5	24.6
3-shot									
w/o compression	0.0	0.0	55.6	0.0	0.0	85.2	0.0	0.0	24.9
5-shot, Setting τ_{code}-0.5									
CODEPROMPTZIP w/o Copy	42.9	41.3	39.8	30.9	28.5	62.4	38.5	49.9	24.1
CODEPROMPTZIP	48.7	45.2	49.9	52.1	45.1	70.9	49.9	41.3	25.1
Oracle	50.0	43.9	55.2	50.0	42.2	74.4	50.0	41.6	25.3
5-shot									
w/o compression	0.0	0.0	57.8	0.0	0.0	86.4	0.0	0.0	25.6

lower τ_{code} of 0.1 achieves the highest quality, approximating complete examples. However, lower τ_{code} values do not always yield better results. For example, in the 1-shot setting on Assertion Generation, $\tau_{code} = 0.3$ achieves an exact match rate of 42.1%, which is lower than the 45.1% obtained with $\tau_{code} = 0.5$ on the same task.

F More Results of Impact of the Number of Shots on the Effectiveness of CODEPROMPTZIP

We also present results under varying shot settings. The findings indicate that increasing the number of compressed examples improves performance, consistent with prior observations for complete examples (He et al., 2024). Notably, this improvement extends to compressed examples. For instance, in Code Suggestion, increasing the number of $\tau_{code} = 0.5$ compressed examples from 1 to 5 raises the CodeBleu score from 22.1 to 25.1, highlighting the advantage of incorporating multiple compressed examples into the prompt.

G Case Study

We present cases across multiple coding datasets, comparing compressed and original code examples. For instance, as demonstrated in Figures 8, 9, and 10, CODEPROMPTZIP prioritizes discarding **Invocation** tokens first, followed by **Symbol** tokens.

```
### FOCAL_METHOD
getProduction(java.lang.String) {
 return productionsByName.get(name); }
### UNIT_TEST
testJustifications() {
 runTest("testJustifications", 2);
      org.jsoar.kernel.Production j =
      agent.getProductions()
      .getProduction("justification-1");
      "<AssertPlaceHolder>";
}
```

Figure 8: Original Code Examples of Assertion Generation (63 tokens)

```
### FOCAL_METHOD
getProduction(java.lang.String) {
 return productionsByName; }
### UNIT_TEST
testJustifications() {
 ;
 org.jsoar.kernel.Production j =
      agent.getProductions()
      .getProduction("justification-1");
      "<AssertPlaceHolder>";
}
```

Figure 9: Compressed Code Examples of Assertion Generation (55 tokens, τ_{code}: 0.1)

```
### FOCAL_METHOD
getProduction(java.lang.String)
 return productionsByName;
### UNIT_TEST
testJustifications()
 ;
 org.jsoar.kernel.Production j = agent;
  "<AssertPlaceHolder>";
```

Figure 10: Compressed Code Examples of Assertion Generation (39 tokens, τ_{code}: 0.4)

```
### BUGGY_CODE
public static TYPE_1
     init(java.lang.String name,
     java.util.Date date) {
   TYPE_1 VAR_1 = new TYPE_1();
   VAR_1.METHOD_1(name);
   java.util.Calendar VAR_2 =
        java.util.Calendar.getInstance();
   VAR_2.METHOD_2(date);
   VAR_1.METHOD_3(VAR_2);
   return VAR_1;
}
### FIXED_CODE
public static TYPE_1
     init(java.lang.String name,
     java.util.Date date) {
   TYPE_1 VAR_1 = new TYPE_1();
   VAR_1.METHOD_1(name);
   java.util.Calendar VAR_2 = null;
   if (date != null) {
       VAR_2 =
            java.util.Calendar.getInstance();
       VAR_2.METHOD_2(date);
   }
   VAR_1.METHOD_3(VAR_2);
   return VAR_1;
}
```

Figure 11: Original Code Examples of Bugs2Fix (195 tokens)

```
### BUGGY_CODE
public static TYPE_1
     init(java.lang.String name,
     java.util.Date date) {
    = new TYPE_1();
   ;
   java.util.Calendar =
        java.util.Calendar;
   .METHOD_2(date);
   .METHOD_3(VAR_2);
   return ;
}
### FIXED_CODE
public static TYPE_1
     init(java.lang.String name,
     java.util.Date date) {
    = new TYPE_1();
   ;
   java.util.Calendar = null;
   if (date != null) {
        = java.util.Calendar;
       .METHOD_2(date);
   }
   .METHOD_3(VAR_2);
   return ;
}
```

Figure 12: Compressed Code Examples of Bugs2Fix (136 tokens, τ_{code}: 0.3)

```
### METHOD_HEADER
protected final void fastPathEmit ( U
     value , boolean delayError ,
     Disposable dispose )
### WHOLE_METHOD
protected final void fastPathEmit(U
     value, boolean delayError,
     Disposable dispose) {
   final Observer<? super V> s = actual;
   final SimplePlainQueue<U> q = queue;
   if (wip.get() == 0 &&
        wip.compareAndSet(0, 1)) {
       accept(s, value);
       if (leave(-1) == 0) {
           return;
       }
   } else {
       q.offer(value);
       if (!enter()) {
           return;
       }
   }
   QueueDrainHelper.drainLoop(q, s,
        delayError, dispose, this);
}
```

Figure 13: Original Code Examples of Code Suggestion (157 tokens, τ_{code}: 0.3)

Original Code Examples (121 tokens, τ_{code}: 0.3)

```
### METHOD_HEADER
protected final void fastPathEmit ( U
     value , boolean delayError ,
     Disposable dispose )
### WHOLE_METHOD
   final Observer<? super V> =
   final SimplePlainQueue<U> =
   if (wip.get() == 0 &&
        wip.compareAndSet(0, 1))
       ;
       if (leave(-1) == 0)
           return;
    else
       .offer(value);
       if (!enter())
           return;
   .drainLoop(q, s, delayError, dispose,
        this);
```

Figure 14: Compressed Code Examples of Code Suggestion (121 tokens, τ_{code}: 0.3)