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

Anonymous ACL submission

Abstract

Retrieval-Augmented Generation (RAG) enhances code generation by incorporating retrieved code examples into prompts, but the resulting long-context inputs impose substantial memory and computational overhead. Existing prompt compression techniques are largely designed for natural language and fail to account for the structural and semantic properties of code, while also lacking fine-grained control over compression ratios. We propose CODEPROMPTZIP, a code-aware prompt compression framework for RAG that enables precise length control while preserving critical information. Motivated by type-aware ablation studies, CODEPROMPTZIP leverages static analysis to rank code tokens by information gain and applies a dynamic compression strategy to retain the most informative tokens under a given budget. For incomplete or unparsable code snippets, CODEPROMPTZIP employs a language-model-based compressor trained on analyzable samples and augmented with a copy mechanism to preserve key tokens. Extensive experiments on three code-related tasks demonstrate that CODEPROMPTZIP consistently outperforms entropy-based and distillation-based baselines, achieving improvements of 23.4%, 28.7%, and 8.7%, respectively, while providing accurate control over compression ratios.

1 Introduction

Retrieval-Augmented Generation (RAG) for language models (LMs) (Lewis et al., 2020; Izacard et al., 2023; Xu et al., 2024) has demonstrated strong performance on knowledge-intensive tasks, especially in the coding domain (Nashid et al., 2023; Chen et al., 2024; He et al., 2024), by incorporating retrieved code examples into input prompts. However, these long-context prompts often span thousands of tokens, posing significant challenges due to substantial memory requirements (e.g., vRAM) and the high cost of processing long inputs with proprietary LMs like GPT-4 (e.g., $2.0 per million tokens).

Prompt compression provides a promising approach to optimize LM utilization by retaining essential information while reducing the prompt length (Chang et al., 2024). Existing research has shown significant success across a variety of 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). However, no compression method has been tailored for coding tasks. Moreover, current approaches lack the ability to offer flexible control over compression ratios without understanding the hierarchical importance of tokens. To address this gap, we propose CODEPROMPTZIP, a type-aware framework designed to compress code examples effectively. By preserving the most informative tokens within a specified length constraint, CODEPROMPTZIP ensures the retention of the necessary information for RAG-based coding tasks while providing precise control over compression ratios.

Firstly, we use static analysis tools to perform ablation analysis on specific token types (e.g., identifiers) in the code prompt, measuring their information gain (IG) (Raileanu and Stoffel, 2004), defined as the increase in predictive uncertainty (i.e., perplexity) in the base LM when a token type is removed. Based on the resulting IG rankings, CODEPROMPTZIP applies a dynamic compression (DC) strategy to generate compressed code examples that recursively preserve the most informative tokens while adhering to a specified length constraint. However, DC cannot be applied directly to the large proportion of incomplete code examples (Yang et al., 2016). For instance, as shown in Table 1, 51% of the code examples in Assertion Generation were unparsable for static analysis, primarily due to reliance on strict syntax rules for token type identification.

To address this limitation, we propose using a small LM (i.e., CodeT5+ (Wang et al., 2023b)) as a compressor to handle unparsable code examples. The LM-based compressor treats a code snippet as a sequence of tokens without being constrained by strict syntax, making the framework applicable to incomplete code. However, there are two main challenges to train the compressor: ❶ lack of compression datasets for coding tasks. Existing approaches often repurpose summarization datasets (Roush and Balaji, 2020) or rely on dataset augmentation using powerful LMs like GPT-4 (Xu et al., 2024). These approaches typically lack diverse compression ratios and fail to account for code-specific characteristics, such as type information. ❷ extractive compression, where the compressed code must be fully derived from the original code.

To address ❶, we leverage parsable examples compressed with the proposed DS approach as the training dataset. This method retains critical tokens under varying ratio constraints, creating a diverse and task-relevant dataset. To address ❷, we enhance the base CodeT5+ architecture with a copy mechanism (See et al., 2017), which introduces a copy distribution over source tokens to guide token preservation during decoding. Furthermore, we extend the vocabulary by incorporating special tokens (e.g., $[Ratio]$), allowing the compressor to condition on specified ratios and adaptively learn to compress code 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 demonstrates improvements over both entropy-based (e.g., Jiang et al. (2023a)) and distillation-based baselines (e.g., 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 of code examples have varying impacts on the retrieval-augmented code generation.
- Based on our observation, we propose a novel prompt compression framework designed for compressing code examples for RAG.
- For unparsable code examples, we developed a copy-enhanced LM as the compressor to compress code examples effectively and allow compression ratio control.
- Through evaluation, we demonstrated that our approach achieves significant improvements over SOTA baselines and demonstrates generalization across different base LMs and tasks. We also make our code public[1].

2 Related Work and Background

2.1 Related work

Prompt compression methods can be classified into two types: **soft prompts** and **discrete prompt compression** (Chang et al., 2024). **Soft prompts** learn embeddings that encode instructions (Mu et al., 2023) or 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 examples into learnable context vectors. However, soft prompts face limitations in cross-model compatibility and require gradient access to base LMs, making them impractical for proprietary LM services.

Discrete prompt compression improves compatibility with black-box or proprietary LMs by selectively retaining informative tokens from the original prompt. Notable works include **entropy-based** and **knowledge distillation** methods. **Entropy-based** approaches, such as Jiang et al. (2023a,b), use small LMs to estimate the information entropy of tokens. However, they evaluate information of prompt tokens by generating that prompt rather than the impact on the task output. **Knowledge distillation** methods use proprietary LMs to produce compressed summaries, which are then employed to train smaller LMs as compressors. For example, Xu et al. (2024) trained a T5 using GPT-3.5 as a teacher model, while Pan et al. (2024) utilized a RoBERTa encoder to classify tokens for extractive compression guided by GPT-4 (Achiam et al., 2023). However, these approaches face challenges in ratio control and involve high costs because of their reliance on proprietary LMs.

Recent work by Yang et al. (2024a) focuses primarily on the natural language components of coding task prompts (e.g., docstrings) rather than on compressing the code itself. However, code exhibits unique characteristics, such as type infor-

[1]https://anonymous.4open.science/r/CodePromptZip-6B2B

mation (Zhang et al., 2024), with different token types encoding distinct symbolic and syntactic information. Code simplification methods such as Zhang et al. (2022) assess token importance based on attention weights; yet these approaches are non-RAG and rely on attention information that is only accessible in open-source base LMs. To the best of our knowledge, we are the first to implement prompt compression for retrieval-augmented code generation.

2.2 Problem Formulation

Referring to Jiang et al. (2023a), we adapt and redefine code prompt compression. For a coding task $\mathcal{T}$, the input prompt is represented as $\boldsymbol{x} = (\boldsymbol{x}_1^{\text{code}}, \ldots, \boldsymbol{x}_N^{\text{code}}, \boldsymbol{x}^{\text{ques}})$, where $\boldsymbol{x}_i^{\text{code}}$ corresponds to the i-th code example, N is the number of examples, and $\boldsymbol{x}^{\text{ques}}$ denotes the associated question. The goal is to compress each code example $\boldsymbol{x}_i^{\text{code}}$ to minimize token usage while preserving essential information.

Formally, the compression process, $Comp()$, can be divided into two cases based on the parsability of the code examples. For parsable code, we apply a static-analysis-based dynamic compression ($\mathcal{DC}$) method, and for unparsable code, we leverage an LM-based compressor ($\mathcal{LM}_{\mathcal{C}}$):

$$\widetilde{\boldsymbol{x}}_i^{\text{code}} = Comp(\mathcal{T}, \tau_{\text{code}}, \boldsymbol{x}_i^{\text{code}}) \quad (1)$$

$$= \begin{cases} \mathcal{DC}(\boldsymbol{x}_i^{\text{code}}, \mathcal{T}), & \text{if } \boldsymbol{x}_i^{\text{code}} \text{ is parsable,} \\ \mathcal{LM}_{\mathcal{C}}(\boldsymbol{x}_i^{\text{code}}, \mathcal{T}), & \text{otherwise.} \end{cases} \quad (2)$$

where $\tau_{code} = 1 - |\widetilde{\boldsymbol{x}}_i^{code}|/|\boldsymbol{x}_i^{code}|$ is the compression ratio for a code snippet, with $\mathcal{T}$ representing specific task contexts to account for variations in token importance across tasks. With compressed code examples, the overall prompt is shortened as:

$$\widetilde{\boldsymbol{x}} = \textsc{CodePromptZip}(\boldsymbol{x}) \quad (3)$$

$$= (\{Comp(\mathcal{T}, \tau_{code}, \boldsymbol{x}_i^{code})\}_{i=1}^{N}, \boldsymbol{x}^{ques})$$

where the overall ratio is given by $\tau = 1 - \widetilde{\boldsymbol{x}}/\boldsymbol{x}$. As code examples account for the majority of token overhead in the prompt, τ_{code} can be regarded as approximately equivalent to τ. The base LM ($\mathcal{BLM}$) using the compressed prompt $\widetilde{\boldsymbol{x}}$ is expected to closely approximate its generation with the original $\boldsymbol{x}$.

3 Motivation: Type-aware Information Rank

Effective prompt compression should prioritize the preservation of the most informative tokens in context (Yang et al., 2024b; Li, 2023). In retrieval-augmented code generation, this raises a key question: *how can we identify which tokens in code examples contribute most to the final generation?*

We employ static-analysis tools to selectively ablate specific token types from the context and measure the resulting change in predictive uncertainty (i.e., perplexity (Kuhn et al., 2023)) in the base LM to estimate their Information Gain (IG) (Raileanu and Stoffel, 2004). The removal of token types that cause higher uncertainty in $\mathcal{BLM}$s indicates that these tokens carry higher IG. Understanding the IG of code tokens thus provides a basic foundation for efficient and informed prompt compression.

3.1 Type Ablation Analysis

We categorize tokens in code examples into five types, following the taxonomy proposed by Wang et al. (2024a):

- **Symbol**: Tokens representing operators, delimiters, and other symbolic elements that define the structure of the code (e.g., `=`, `{`, `;`, `,`).
- **Signature**: Tokens defining the declaration and parameters of methods (e.g., `calculate(int x)`).
- **Invocation**: Tokens related to function or method calls, capturing interactions and dependencies within the code.
- **Identifier**: Tokens that serve as variable names, class names, or other user-defined labels.
- **Structure**: Tokens associated with loops, conditionals, and other flow control statements, which dictate the logical behavior of the program (e.g., `if`, `for`, `while`).

We construct Abstract Syntax Trees (ASTs) using JavaParser (JavaParser, 2019) to identify token types. For each type, we remove all corresponding tokens from the retrieved code examples and then perform RAG with the type-ablated examples to measure their impact on $\mathcal{BLM}$s' predictive uncertainty. The IG of type C is defined as its effect on

perplexity:

$$IG(C) = \frac{1}{|C|}\left(\log \text{PPL}(y \mid \boldsymbol{x} \setminus C) - \log \text{PPL}(y \mid \boldsymbol{x})\right) \tag{4}$$

where $\text{PPL}(y \mid \mathbf{x} \setminus C)$ denotes the perplexity of the base LM's output conditioned on the type-ablated prompt $\mathbf{x} \setminus C$. We normalize IG by the number of tokens ablated $|C|$ of the ablated type to enable a fair comparison across types of different sizes. We compute PPL as

$$\text{PPL}(y \mid \boldsymbol{x}) = \exp\left(-\frac{1}{|y|}\sum_{j=1}^{|y|}\log P(y_j \mid y_{<j}, \mathbf{x})\right) \tag{5}$$

The defination of IG indicates that the absence of a type C that leads to higher perplexity increases the model's uncertainty, implying that such tokens are more informative.

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.

Dataset: (i) **Assertion Generation** (Nashid et al., 2023): The input is a focal method (the method under test) and its partial unit test, while the outputs are assertion statements verifying its correctness. (ii) **Bugs2Fix** (Lu et al., 2021): The input is a buggy method, and the output is a refined version of the method with the bugs fixed. (iii) **Code Suggestion** (Chen et al., 2024): 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.

Task	Knowledge Base (Parsable)	Test	Val
Assertion	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 1), and craft task-specific instructions in a one-shot setting.

As listed in Table 1, we follow the original split of the dataset into Training, Validation, and Test partitions. The training partition functions as our knowledge base for retrieval. Note that some code examples that yield parsing errors in JavaParser due to code incompleteness are classified as **Unparsable**. Due to resource constraints, we randomly sample 2,000 instances from both the validation and test sets for this ablation analysis. Queries from the sampled validation set are used to study example importance, while queries from the sampled test set are used to evaluate compression performance in the remainder of the paper.

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 :

(a) Assertion Generation

Demonstrations:
[START]
BUGGY_CODE:
{*buggy method*}

FIXED_CODE:
{*repaired method*}

...
[END]
Query
[START]
BUGGY_CODE:
{*buggy method*}

FIXED_CODE :

(b) Bugs2Fix

Demonstrations:
[START]
METHOD_HEADER:
{*header*}

WHOLE_METHOD:
{*body*}

...
[END]
Query
[START]
METHOD_HEADER:
{*header*}

WHOLE_METHOD:

(c) Code Suggestion

Figure 1: The illustration of different RAG coding tasks alongside their respective prompt templates.

Base LMs: The in-context learning capabilities of $\mathcal{BLM}$s enable them to exploit query-related examples for more instruction-aligned outputs. To assess whether token-type influence remains consistent across models, we evaluated our prompts on GPT-4o-mini and CodeLlama-13B with temperature set to 0 for stable results. We also tested CODEPROMPTZIP on Gemini-1.0-Pro in the remainder experiment, but perplexity could not be computed on this $\mathcal{BLM}$.

Retrievers: According to empirical studies (He et al., 2024; Wang et al., 2024b), BM25 is recognized as a SOTA open-source and unsupervised retriever for coding tasks. Moreover, BM25 is commonly utilized in baseline methods such as Jiang et al. (2023b), Pan et al. (2024), and Xu et al. (2024). Therefore, we also adopted the BM25 for retrieving examples.

3.3 Observation

Figure 2 shows the IG rankings of token types in descending order. The more informative types should be prioritized for retention during compression. Notably, the hierarchical order remains consistent across different $\mathcal{BLM}$s, underscoring the cross-model transferability of type-aware compression.

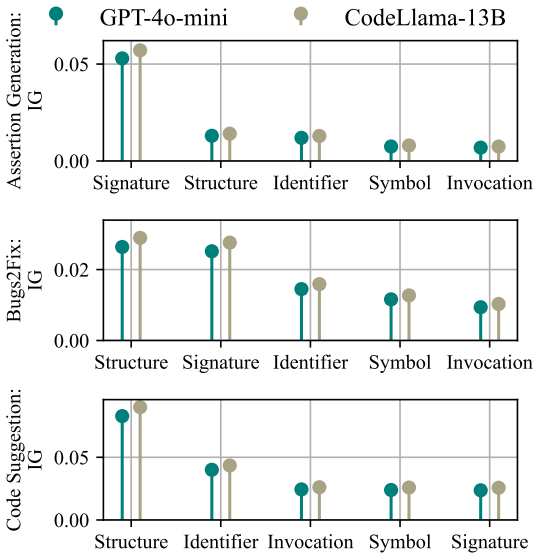


Figure 2: IG of code token types: e.g., Symbol > Invocation in Assertion Generation, and vice versa in Code Suggestion. IGs are task-specific yet model-agnostic, applicable to both GPT-4o-mini and CodeLlama-13B.

4 Methodology

As shown in Figure 3, CODEPROMPTZIP operates in two distinct phases. In the training phase, given a type-aware IG ranking for a task $\mathcal{T}$, we apply a dynamic compression to recursively preserve tokens with informative types. This process transforms $(\boldsymbol{x}_i^{code}, \tau_{code}, \mathcal{T})$ into $\widetilde{\boldsymbol{x}}_i^{code}$.

Although Algorithm 1 can directly generate compressed code, its implementation relies on JavaParser-based token labeling, limiting its applicability to unparsable code in the knowledge base. However, as shown in Table 1, unparsable code examples are prevalent in coding tasks.

To enhance **applicability** of CODEPROMPTZIP, we train $\mathcal{LM}_\mathcal{C}$ on a dataset constructed from parsable examples. By processing code sequences as probabilistic relations (Xu et al., 2024; Pan et al., 2024) rather than strictly adhering to exact syntax, $\mathcal{LM}_\mathcal{C}$ 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 and the retrieved code examples, the handling of code examples depends on their parsability. Parsable code examples are processed using dynamic compression, while unparsable ones are handled by seq-to-seq $\mathcal{LM}_\mathcal{C}$. The compressed examples are then aggregated into a prompt and passed to $\mathcal{BLM}s$ to produce the final output.

Algorithm 1 Dynamic Compression

Input: Original code snippet with L tokens $\boldsymbol{x}_i^{code} = \{x_j\}_{j=1}^{L}$, and IG $\{v_1, v_2, \ldots, v_L\}$, and a capacity $W = \tau_{code} \cdot L$.
Output: Compressed Code $\widetilde{\boldsymbol{x}}_i^{code}$.

Initialize a dynamic programming table dp of size $(L+1) \times (W+1)$ with all zeros.
for $i = 1$ to L **do**
  $dp[i][0] \leftarrow 0$
  for $w = 0$ to W **do**
    $dp[i][w] \leftarrow \max(dp[i-1][w], dp[i-1][w-1] + v_i)$
  end for
end for
Initialize an empty set *selectedTokens*.
$w \leftarrow W$.
for $i = L$ to 1 **do**
  if $dp[i][w] \neq dp[i-1][w]$ **then**
    Add token i to *selectedTokens*.
    $w \leftarrow w - 1$.
  end if
end for
return $\widetilde{\boldsymbol{x}}_i^{code} \leftarrow$ *selectedTokens*.

4.1 Dynamic Compression of Parsable Code Examples

Code example compression can be modeled as a **0-1 knapsack** (Zhang et al., 2022): the token capacity is $W = \tau_{code} \cdot L$, and the original code snippet has L tokens. Each token has the unit weight and a corresponding IG (i.e., contribution to the final output). We can either include a token in the "knapsack" or discard it. The goal is to preserve tokens selectively to maximize the total information gain.

We derive the type-aware information ranking in the Motivation section. For in-type tokens, their values are determined by their frequency, as high-frequency tokens carry less information, following information theory (Shannon, 2001). For example, in Assertion Generation, we reassign integer values, ranging from 1 to 5, to represent the five token types for easy ranking. Among these, signature tokens are assigned the highest type value of 5. To incorporate frequency, if a signature token appears with a frequency of 0.2, its value is calculated as the type value (5) plus the complement of its frequency (1 - 0.2 = 0.8), resulting in a value of 5.8. Conversely, another signature token with a higher frequency of 0.8 would have a lower overall value of 5.2. This approach enables a hierarchical ranking that prioritizes tokens of informative types while accounting for their token frequency.

To avoid prematurely discarding critical tokens, tokens that belong to multiple types (e.g., method signature tokens always include (), which serve as both symbol and signature tokens.) are assigned

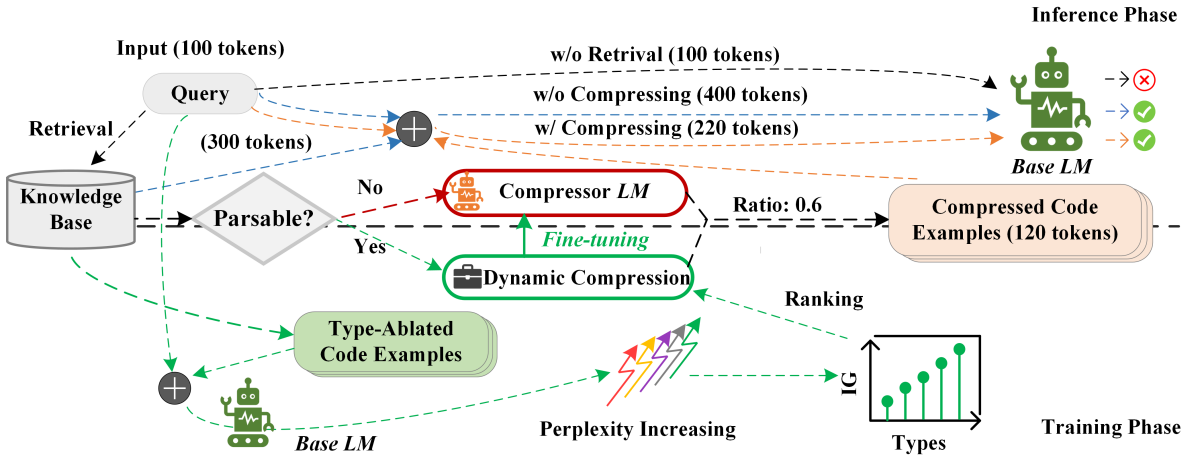


Figure 3: Framework of CODEPROMPTZIP.

the highest value among their categories. For out-of-type tokens (e.g., return), which do not belong to any predefined category, we assign them a value of max(values) + 1. This ensures that these tokens are treated as the most critical and removed last during the compression process.

To address the knapsack for code compression, we adopt a dynamic compression approach (Algorithm 1) using dynamic programming (dp) (Pferschy, 1999) as follows: Initialization (Line 1): A dynamic programming (dp) table is created to store all intermediate solutions. $dp[i][w]$ represents the maximum information that can be obtained using the first i tokens within a capacity w. Recursive Calculation (Lines 2–7): For each token i and capacity w. The algorithm computes the recursive solution by determining whether to include or exclude the token. If included, $dp[i][w] = dp[i-1][w-1]+v_i$. If excluded, $dp[i][w] = dp[i-1][w]$. The optimal subsolution is the maximum of these two choices. Backtracking to Select Tokens (Lines 8–13): An empty set $selectedTokens$ is initialized to track the preserved tokens. Starting from the last token, the algorithm includes the token i if it contributes to the maximum value, updating $selectedTokens$ accordingly. Output: The compressed code example is constructed from the $selectedTokens$.

4.2 $\mathcal{LM}_\mathcal{C}$-based Compression of Unparsable Code Examples

To effectively compress unparsable code examples, we adopt CodeT5+ (Wang et al., 2023a) as our base model and introduce two key architectural modifications. First, we extend the input vocabulary with task-indicative tokens such as *[ASSERTION]*, *[BUGS2FIX]*, and *[SUGGESTION]*, which are prepended to the input sequence to explicitly specify the task context. Second, to enable $\mathcal{LM}_\mathcal{C}$ to condition on flexible compression ratios τ_{code}, we introduce special tokens *[Ratio]*, *[/Ratio]*, *[Compress]*, and *[/Compress]*. These tokens guide the compressor to generate code snippets tailored to the given compression ratio and task requirements.

In addition, we integrate a copy mechanism (See et al., 2017; Zhang et al., 2021) into the model architecture. This mechanism allows the model to directly copy tokens from the input sequence, which aligns with the extractive nature of code compression where outputs are derived largely from the original code. Notably, these modifications are general and can be applied to both encoder–decoder and decoder-only transformer architectures beyond CodeT5+.

We leverage cross-attention to estimate the copy probability of source tokens. Specifically, attention weights α are computed as:

$$\alpha = \text{softmax}\left(\frac{Q_{dec}K_{enc}^{\top}}{\sqrt{d}}\right) \quad (6)$$

where Q_{dec} and K_{enc} are the query and key matrices of the decoder and encoder, and d is the hidden dimensionality. The resulting α guides the decoder to focus on salient input tokens. The context vector h^* is then computed as:

$$h^* = \alpha \cdot V_{enc} \quad (7)$$

where V_{enc} is the encoder's value matrix. This context vector dynamically summarizes relevant input content at each decoding step. To decide between copying and generating, we concatenate h^* with the decoder input x_{dec} and feed the result

into a linear copy layer to compute the generation probability p_{gen}:

$$p_{gen} = \sigma(W_{gen}[h^*; x_{dec}] + b_{gen}) \quad (8)$$

where σ is the sigmoid function, and W_{gen}, b_{gen} are learnable parameters. Here, p_{gen} denotes the probability of generating a token from the vocabulary, while $1 - p_{gen}$ denotes the probability of copying a token from the source using α.

The final output distribution is then computed as:

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

where P_{vocab} is the vocabulary distribution and P_{copy} corresponds to the attention-based copy distribution. This mechanism not only directs the decoder's focus to informative tokens but also enhances robustness in handling rare or domain-specific identifiers.

The compressor is trained using the standard cross-entropy loss:

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

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

To construct fine-tuning data, we apply dynamic compression to parsable code examples in the knowledge base, generating compressed snippets at varying ratios (τ_{code} from 0.1 to 0.9). Each example is paired with its compressed counterpart, and the dataset is split into training, validation, and test sets in an 8:1:1 ratio.

5 Experimental Setting

5.1 Baselines

We compare our approach with four state-of-the-art prompt compression baselines from two major families, as introduced in the Related Work section. 2.1: **(1) Entropy-based methods** - LLMLingua (Jiang et al., 2023a) and LongLLMLingua (Jiang et al., 2023b); and **(2) Knowledge-distillation methods** - LLMLingua-2 (Pan et al., 2024) and RECOMP (Xu et al., 2024).

5.2 Datasets and Metrics

We evaluated the performance of CODEPROMPTZIP on the same three coding tasks described in the Motivation section with the same prompt template. Assertion Generation is measured by the Exact Match rate. Bugs2Fix is evaluated using CodeBLEU (Ren et al., 2020), consistent with the CodexGLUE benchmark (Lu et al., 2021). Code Suggestion is also evaluated using CodeBLEU.

5.3 Base LMs

CODEPROMPTZIP, a discrete prompt compression approach, offers the advantage of generalization across different $\mathcal{BLM}$s (Xu et al., 2024; Jung and Kim, 2024). Although our main experiments are conducted on GPT-4o-mini, we further evaluate its generalization by testing on two additional $\mathcal{BLM}$s: CodeLlama-13B (Roziere et al., 2023) and the proprietary Gemini-1.0-pro (Team Gemini et al., 2024).

5.4 Implementation Details

To ensure consistent outputs, we set the temperature and top-p of $\mathcal{BLM}s$ to 0. The $\mathcal{LM}_{\mathcal{C}}$ is trained using the AdamW optimizer with a batch size of 16, a learning rate of 5×10^{-5}, and 1,000 warmup steps over 10 epochs. All other settings follow the default CodeT5+ configuration. All experiments were conducted on a Linux server equipped with four Nvidia RTX 3090 GPUs and 24 CPU cores.

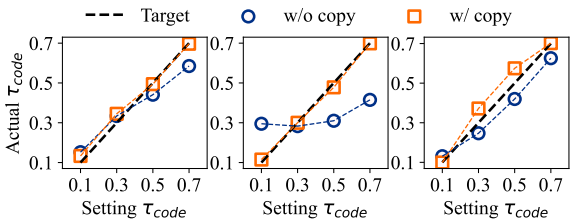
6 Results

6.1 Comparisons with NL-specific Baselines

CODEPROMPTZIP consistently outperforms baselines across the studied coding tasks. 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) or removing random tokens perform 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. Although uncompressed prompts achieve the highest quality metrics overall, they come at a substantial token cost.

Approach	Assertion Generation # tokens	τ(%)	Exact Match(%)	Bugs2Fix # tokens	τ(%)	CodeBleu(%)	Code Suggestion # tokens	τ(%)	CodeBleu(%)
w/o retrieval	334	46.6	23.9	122	66.3	41.7	29	82.6	14.2
Random	440	29.7	25.2	262	27.4	40.3	121	27.5	16.8
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
w/o Compression	626	0.0	50.5	362	0.0	81.4	167	0.0	24.7

Table 2: Results on three coding tasks using GPT-4o-mini 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.



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

Figure 4: Compression ratio control.

6.2 Compression Ratio Control

Our dynamic compression strictly enforces length constraints, while the LM-based compressor leverages an extended vocabulary and accepts τ_{code} as input, enabling adaptive compression of unparsable code examples to meet the desired ratio. As shown in Figure 4, the specified τ_{code} closely matches the actual achieved values. The dotted line (Target) indicates the ideal outcome, and CODEPROMPTZIP (w/ copy) aligns closely with this benchmark.

In contrast, baselines such as LLMLingua2 struggle with ratio control. Since prompts distillated from GPT-4 always ignores length constraints (Pan et al., 2024), the generated examples often deviate significantly from the target. As reported in Table 2, when configured for a ratio of 0.3, LLMLingua2 produced examples with an actual ratio of 0.251.

6.3 Ablation Study of Copy Mechanism

The copy module in $\mathcal{LM}_{\mathcal{C}}$ introduces a copy distribution over source tokens, enabling the model to directly preserve tokens from the input sequence instead of generating them from the entire vocabulary. This mechanism significantly benefits the extractive learning of compression, improving both effectiveness and ratio control.

As shown in Tables 2, the ablation study (*w/o copy*) underscores the consistent contributions of the copy mechanism to enhancing $\mathcal{LM}_{\mathcal{C}}$. For instance, integrating the copy mechanism improves Exact Match by 1.2% for Assertion Generation and boosts CodeBLEU by 5.2% for Bugs2Fix and 3.2% for Code Suggestion, compared to the $\mathcal{LM}_{\mathcal{C}}$ of original CodeT5+.

In terms of ratio control, Figure 4 demonstrates that compressors based on the original CodeT5+ architecture struggle to generate outputs that align with the desired compression ratio. This results in outputs that often deviate from specified ratio settings and yield lower effectiveness.

7 Conclusion

This paper presents CODEPROMPTZIP, a novel framework to compress retrieved code examples before integrating them into prompts. First, we proposed an approach to compress parsable examples via dynamic compression. This method preserves informative tokens across different compression ratios while maintaining code semantics. Moreover, for unparsable code examples, we develop a copy augmented CodeT5+ as a compressor, allowing the compressor to directly select and preserve essential tokens from the original code example end-to-end. We also introduced special tokens (e.g., [Ratio]) to enable ratio-conditioned compression. Experimental results on three coding tasks demonstrate that CODEPROMPTZIP improves the efficiency of retrieval-augmented LMs while maintaining minimal performance degradation and consistently outperforms SOTA baselines, achieving improvements of 23.4%, 28.7%, and 8.7%, respectively. Furthermore, CODEPROMPTZIP provides flexible control of compression ratio, which is important in practice. 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 $\mathcal{LM}_C$ 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 importance of tokens in our approach depends 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 information gain of different types may differ. Although the token value is easily adjusted in dynamic compression, additional training of the compressor LM is required; our framework can be easily applied to any downstream tasks. Furthermore, as shown in Figure 1 of the original paper, while information rankings are task-specific, certain patterns emerge consistently. For example, **Structure** tokens exhibit a higher value than **Identifier** tokens across all tasks. We also show the cross-domain capability of CODEPROMPTZIP by a cross-task experiment and even in such a setting, our approaches still outperform the baselines in most of the cases. More importantly, we develop a framework that enables us to optimize the compressor for downstream tasks, thereby achieving more effective and task-aware compression.

8.1 Threats to validity

Threats to external validity This study focuses exclusively on Java and three coding tasks. Our findings might not be generalized to other programming languages and tasks. However, our approach can be easily replicated on other programming languages like Python also have static-analysis tools and other tasks. Future research could extend this work to other tasks and languages. Our experiments utilized GPT-4o-mini, Gemini, and CodeLlama-13b. We encourage further studies to explore additional base LMs and a broader range of programming languages and coding tasks.

Threats to internal validity The ranking of type information is derived from the entirety of the training dataset, making it highly dependent on the representativeness and diversity of the training samples. If the training dataset lacks sufficient variety, the resulting token information rankings may fail to generalize effectively to unseen examples. Consequently, for specific code examples, the generalized overall ranking may lead to suboptimal token removal strategies, thereby adversely affecting generation effectiveness.

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*.

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.

JavaParser. 2019. Javaparser. `https://github.com/javaparser/javaparser`.

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.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. *Preprint*, arXiv:2302.09664.

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.

Ulrich Pferschy. 1999. Dynamic programming revisited: Improving knapsack algorithms. *Computing*, 63(4):419–430.

Laura Elena Raileanu and Kilian Stoffel. 2004. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93.

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*.

Allen Roush and Arvind Balaji. 2020. Debatesum: A large-scale argument mining and summarization dataset. In *Proceedings of the 7th Workshop on Argument Mining*, pages 1–7.

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.

Claude Elwood Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55.

Gemini Team Gemini, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2024. 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. 2024a. 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, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023a. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *Preprint*, arXiv:2305.07922.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024b. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*.

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).

Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 391–402.

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.

Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1073–1084.

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.

A Appendix

A.1 Qualitative Analysis

To demonstrate why CODEPROMPTZIP outperforms the baselines, we present a case study from Bugs2Fix comparing compressed code examples generated by CODEPROMPTZIP and LLMLingua2 (Pan et al., 2024). Figure 5 presents the original code example. The bug is fixed by adding an if statement to check if data is null or not (i.e., if(date != null)). Figure 6 and Figure 7 present the compressed code example after being compressed by our and LLMLingua-2. CODEPROMPTZIP effectively retains **Signature** tokens, which are proved important in Bugs2Fix (see Figure 1 of the original paper). In contrast, LLMLingua2 discards these critical tokens and does not really demonstrate how the bug is fixed. Additionally, LLMLingua2 faces challenges in ratio control; for example, when configured for a ratio of 0.3, it produces examples with a ratio closer to 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 5: Original Code Examples of Bugs2Fix (195 tokens)

A.2 Generalization with Different $\mathcal{BLM}$

CODEPROMPTZIP consistently outperforms baselines across studied $\mathcal{BLM}s$ CodeLlama-13B and Gemini-1.0. Fig. 8 compares the performance among the studied prompt compression approaches across two additional $\mathcal{BLM}s$ and all three tasks. CODEPROMPTZIP consistently outperforms baselines across the studied $\mathcal{BLM}s$. For instance, CODEPROMPTZIP outperforms the best baselines by 14.3%, 28.1%, and 9.0% for Assertion Generation, Bugs2Fix, and Code Suggestion using Gemini-1.0-pro, respectively. The consistent superiority highlights the robustness and effectiveness of CODEPROMPTZIP as a transferable and generalized compression approach for code-related tasks, attributed to its model-agnostic information ranking.

```
### 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 6: Compressed Code Examples by CODEPROMPTZIP of LLmlingua2 (130 tokens, τ_{code}: 0.3)

```
### BUGGY CODE
TYPE _ 1 init
1 1 TYPE _ 1
METHOD _ 1
2
getInstance
2 METHOD _ 2
1 METHOD _ 3 2
return _ 1
### FIXED_CODE
TYPE 1 init
_ 1 VAR 1 TYPE _ 1
METHOD _ 1
2 null
VAR _ 2 getInstance
2 METHOD _ 2 date
VAR _ 1. METHOD _ 3 ( VAR 2
return VAR _ 1
```

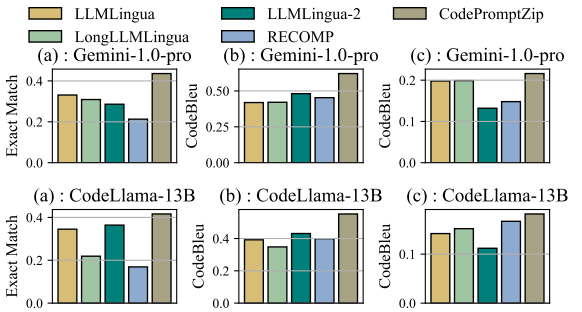
Figure 7: Compressed Code Examples by LLMLingua2 of Bugs2Fix (114 tokens, τ_{code}: 0.4)

A.3 Impact of τ_{code} and Shot numbers and their trade-off

In Figure 9, we illustrate the impact of τ_{code} and the number of shots on the quality of the genera-

Table 3: Cross-task Results: **Bold** font indicates the in-task scenario.

Compressor			(a): Assertion Generation		(b): Bugs2Fix		(c): Code Suggestion	
(a)	(b)	(c)	τ_{code}(%)	**Exact Match(%)**	τ_{code}(%)	**CodeBleu(%)**	τ_{code}(%)	**CodeBleu(%)**
✓			**31.5**	**42.1**	30.9	57.0	29.6	18.9
	✓		27.8	41.9	**30.0**	**61.9**	28.1	21.8
		✓	32.1	38.5	31.4	52.8	**32.2**	**23.7**



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

Figure 8: Performance of the proposed CODEPROMPTZIP across different $\mathcal{BLM}$s.

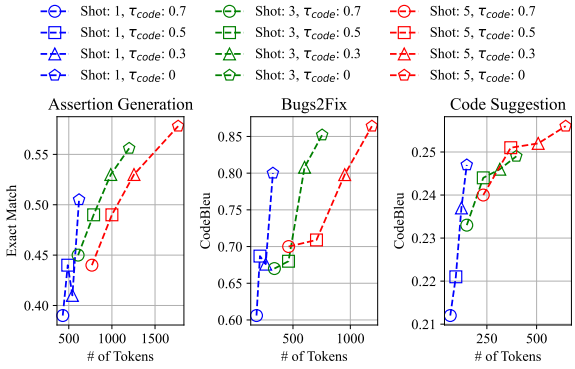


Figure 9: Trade-off between keeping more tokens in a single example or including more examples.

tion. Under fixed-shot settings, performance varies with different τ_{code}. Lower τ_{code} values generally yield better results. Increasing the number of shots typically improves performance as well, though exceptions exist.

When comprehensively considering the trade-off between these factors, the situation becomes more complex, raising an important question: **Given a fixed token budget, how should we balance the number of examples (shots) and the degree of code compression (τ_{code})?** Specifically, should the prompt include fewer, less-compressed (i.e., more complete) examples, or a greater number of highly compressed ones? This critical issue remains underexplored in existing approaches (Jiang et al., 2023a; Xu et al., 2024), which predominantly rely on fixed-shot settings.

In general, using fewer examples with more tokens allocated to each example tends to yield better performance than using more examples with aggressive compression. In particular, high compression ratios often degrade performance relative to using fewer, uncompressed examples. For instance, in Assertion Generation, with a 700-token budget, using a single uncompressed example (i.e., *1-shot* with $\tau_{code} = 0$) achieves a higher Exact Match rate, outperforming settings that include three highly compressed examples at $\tau_{code} = 0.5$ and 0.7.

However, this does not imply that compression is ineffective. The optimal trade-off depends on the specific token budget. When the token budget increases to between 1000, the dynamics change. Although the budget allows for one full example, it is insufficient to include three complete examples. In this case, compression becomes beneficial, enabling the inclusion of three moderately compressed examples (i.e., *shot 3*, $\tau_{code} = 0.5$), which results in improved performance.

A.4 Out-of-domain Capability

To evaluate out-of-domain effectiveness of CODEPROMPTZIP, we conducted cross-task experiments. Specifically, for each downstream task serving as the new task, in-task rankings were applied in dynamic compression to handle parsable code examples, unparsable code snippets were processed by $\mathcal{LM}_{\mathcal{C}}$, which was trained on other coding tasks. Notably, task-specific special tokens (e.g., [ASSERTION]) were not used in these experiments. Table 3 summarizes the results of these cross-task evaluations.

Although the effectiveness of CODEPROMPTZIP slightly degrades in cross-task scenarios, it still outperforms baselines in most cases. For instance, in the first column of Table 3, Assertion Generation applies $\mathcal{LM}_{\mathcal{C}}$ trained on Bugs2Fix and Code Suggestion to compress unparsable code examples. The Bugs2Fix in-task compressor achieves an Exact Match rate of 41.9% with a τ_{code} of 27.8% on Assertion Generation, compared to its in-task performance of 42.1%

Exact Match rate and a τ_{code} of 31.5%. Despite a slight drop in precision for the desired ratio and performance, CODEPROMPTZIP remains robust, outperforming baselines like LongLLMlingua, which achieves an Exact Match rate of 34.1%.

This observation highlights an inherent trade-off between generalizability and task-specific effectiveness. Compressors trained on diverse tasks may capture broader code transformation patterns, enabling them to generalize reasonably well to new domains. However, they may lack the nuanced understanding needed to fully optimize for any single target task. In contrast, in-task compressors are better attuned to domain-specific syntax and semantics, often yielding slightly better performance when applied within the same task. In this study, we develop a framework that enables us to optimize the compressor for downstream tasks, thereby achieving more effective and task-aware compression.