Enhanced Prompting Framework for Code Summarization with Large Language Models

MINYING FANG*, Qingdao University of Science and Technology, China
XING YUAN*, Qingdao University of Science and Technology, China
YUYING LI†, Qingdao University of Science and Technology, China
HAOJIE LI, Qingdao University of Science and Technology, China
CHUNRONG FANG, Nanjing University, China
JUNWEI DU†, Qingdao University of Science and Technology, China

Code summarization is essential for enhancing the efficiency of software development, enabling developers to swiftly comprehend and maintain software projects. Recent efforts utilizing large language models for generating precise code summaries have shown promising performance, primarily due to their advanced generative capabilities. LLMs that employ continuous prompting techniques can explore a broader problem space, potentially unlocking greater capabilities. However, they also present specific challenges, particularly in aligning with task-specific situations—a strength of discrete prompts. Additionally, the inherent differences between programming languages and natural languages can complicate comprehension for LLMs, impacting the accuracy and relevance of the summaries in complex programming scenarios. These challenges may result in outputs that do not align with actual task needs, underscoring the necessity for further research to enhance the effectiveness of LLMs in code summarization.

To overcome these limitations, we combine the strengths of the two approaches described above and introduce EP4CS—an Enhanced Prompting framework for Code Summarization with Large Language Models. Firstly, we design Mapper, which undergoes pre-training on <Code, Knowledge> pairs and facilitates the optimization and updating of prompt vectors based on the outputs of LLMs. Additionally, we develop a Struct-Agent that enables LLMs to more accurately interpret the complex code by in-depth analysis of the programming language's syntax and structure. Experimental results indicate that, compared to existing baseline methods, our enhanced prompting learning framework significantly improves performance while maintaining the same parameter scale. Specifically, when evaluated on Java using StarCoderBase$_{1B}$, EP4CS achieved score improvements of 6.59% on BLEU, 7.06% on METEOR, and 4.43% on ROUGE-L, while also demonstrating strong robustness. And it's closer to real-world scenarios in terms of semantic metrics SentenceBERT. The results from the human evaluation and case studies show that EP4CS surpasses the baseline methods, producing higher-quality and more relevant summaries.

CCS Concepts: • **Software and its engineering → Software development techniques**.

Additional Key Words and Phrases: Source Code Summarization, Large Language Model, Prompt Learning

*These authors contributed equally to this work.
†Both authors are corresponding authors.

Authors' Contact Information: Minying Fang, Qingdao University of Science and Technology, Qingdao, Shandong, China, fangmy@mails.qust.edu.cn; Xing Yuan, Qingdao University of Science and Technology, Qingdao, Shandong, China, 4023112033@mails.qust.edu.cn; Yuying Li, Qingdao University of Science and Technology, Qingdao, Shandong, China, liyy@qust.edu.cn; Haojie Li, Qingdao University of Science and Technology, Qingdao, Shandong, China, lihaojie@qust.edu.cn; Chunrong Fang, Nanjing University, Nanjing, Jiangsu, China, fangchunrong@nju.edu.cn; Junwei Du, Qingdao University of Science and Technology, Qingdao, Shandong, China, dujunwei@qust.edu.cn.

ACM 2994-970X/2025/7-ARTISSTA072
https://doi.org/10.1145/3728949

ACM Reference Format:
Minying Fang, Xing Yuan, Yuying Li, Haojie Li, Chunrong Fang, and Junwei Du. 2025. Enhanced Prompting Framework for Code Summarization with Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA072 (July 2025), 24 pages. https://doi.org/10.1145/3728949

1 Introduction

The widespread lack of sufficient code summarization poses significant challenges to the long-term maintainability of software, increasing the likelihood of costly errors and delays [10, 51, 54]. At the same time, studies show that approximately 25% of comments become outdated or invalid during a project's lifecycle, adding to the complexity of maintenance [43]. However, the manual creation of code summaries is a time-consuming and tedious task. Many developers, under the pressure of tight development cycles and deadlines, often choose to write brief summaries or even forgo them entirely [64]. Every developer experiences the frustration of inheriting poorly documented code, spending hours deciphering its intent rather than building innovative features [24]. Given the increasing complexity of software systems and the accelerated development cycles, automated solutions are no longer just a convenience but a necessary condition for ensuring efficient and error-free code management [24, 43, 51].

The advent of LLMs has recently revolutionized this task, demonstrating increasingly powerful generative capabilities [15, 25, 42, 46, 49]. A growing body of research is focusing on integrating LLMs into software engineering tasks [22, 31], and this integration is poised to transform the development process by significantly enhancing both automation and precision [27, 47, 48]. For instance, InferFix [29] trains LLMs on specialized defect and repair datasets, leveraging static analysis tools to autonomously identify and resolve software defects, improving efficiency and accuracy in defect management. Bhattacharya et al. [6] demonstrate the potential of fine-tuning various LLMs to generate summaries of code snippets based on precise instructions. Although these method are highly effective, they come with significant challenges, including high training costs and potential security vulnerabilities. Modifications in model parameters could inadvertently undermine original human supervision, highlighting the need for further research on cost-effective training strategies and secure model adaptation.

As shown in Fig. 1 (a), prompt learning methods can significantly enhance the precision and expressiveness of LLM in task understanding and generation. To be specific, in-context learning [11] draws on solutions that mirror existing tasks, facilitating analogical reasoning. Research [17] demonstrates that CodeX has successfully generated code summaries through few-shot examples, enhancing the model's ability to comprehend human intent. Furthermore, studies [51, 54, 55] highlight the effectiveness of leveraging historical databases to create contextual frameworks, enabling LLMs to address requests with multiple, concurrent intentions. Additionally, [61] has been shown to significantly boost LLMs' reasoning capabilities, prompting the models to systematically analyze the logic behind code, check error outputs, and identify potential root causes, instead of relying on guesswork. As shown in Fig. 1(b), PromptCS [53] enhances LLMs by integrating external 'soft prompt' vectors directly, eliminating the need for manual construction of discrete prompts. This eliminates the need for manually constructing task-specific prompts and allows the model to quickly adapt to new tasks and datasets.

Despite these advancements, the above methods exhibit certain deficiencies in code summarization tasks, presenting avenues for further research and refinement [15, 25]. Firstly, although the continuous prompt method [38, 39, 53] avoids the design of complex textual instructions, it cannot provide the background knowledge and expert experience relevant to LLM tasks as the discrete prompt method does. It cannot enhance LLMs' problem-solving capabilities by connecting new queries with previously solved, similar tasks. Secondly, previous researches [2, 9] have used

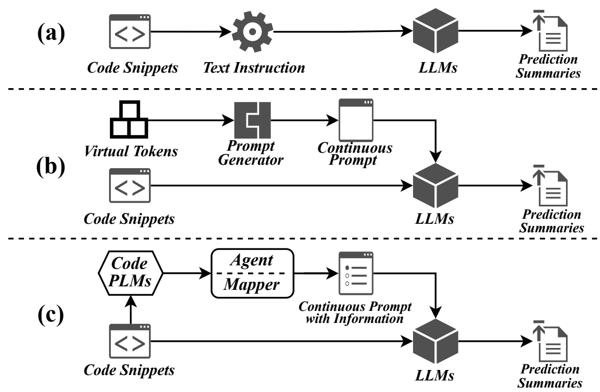


Fig. 1. (a) Using discrete prompt methods (eg., zero-shot, few-shot, in-context learning, the chain of thought), LLMs can acquire expert knowledge and generate summaries with well-designed templates; (b) ***PromptCS*** utilizes continuous prompt, circumvents the complex instruction design required in (a), though it lacks detailed knowledge integration; (c) We design the ***Mapper*** and ***Struct-Agent*** to amalgamate the strengths of both (a) and (b), thereby guiding LLMs more effectively.

compilers and parsers to convert code into structured forms, such as abstract syntax tree and control flow graph to enhance the LLMs' understanding of programming languages. However, this form of knowledge input is suboptimal due to the modal differences between natural languages and programming languages. The absence of background knowledge and code structure semantics not only reduces the learning efficiency of the model in the initial phase but also leads to the generation of content that is too general and does not meet the requirements of practical applications.

To address the challenges mentioned, we propose an **E**nhanced **P**rompting framework **f**or **C**ode **S**ummarization with Large Language Models(EP4CS). Specifically, EP4CS incorporates a two-stage training process and consists of four main components: ***Code Encoder***, ***Mapper***, ***Struct-Agent***, and ***LLMs***. In the first stage, the *Mapper* undergoes bimodal pre-training under multiple supervised tasks with <Code, Knowledge> pairs, allowing its internal virtual tokens to simulate task-relevant background knowledge. During the second stage, code snippets are processed by *Mapper* and *Struct-Agent* to generate knowledge-enhanced prompts and structured prompts, which are then aligned with the intrinsic embedding distribution of the large model. Meanwhile, the LLMs, with the parameters frozen, dynamically generate code summaries in response to the prompts. Finally, by dynamically adjusting the continuous prompt vectors and other trainable modules through backpropagation, the framework's performance and generalization capabilities are enhanced.

In summary, we make the following contributions:

- We propose a prompt learning framework called **EP4CS**, which generates knowledge-enhanced prompt vectors through a *Mapper* module, thereby improving the ability of large language models to produce high-quality code summaries.
- We design *Struct-Agent*, a module that extracts structural semantics from code and maps them into a high-dimensional latent space, effectively bridging the gap between code structure and LLMs' comprehension capabilities.
- We conduct validation on a wide range of programming language datasets, and the experimental results show that EP4CS significantly outperforms existing frameworks under four indicators and is closer to or even surpasses task-oriented fine-tuning schemes. And it's a general framework and can be combined with LLMs.

2 Background and Related Works

2.1 Code Summarization

In early research on code summarization, researchers primarily rely on information retrieval (IR) techniques to match keywords in a database that are similar to the source code and synthesize them into new summaries [12, 21, 62]. Then, most neural approaches to source code summarization frame the problem as a sequence generation task. The earliest work using RNN [56] networks with attention mechanisms to generate summaries for source code snippets. Allamanis et al. [3] develops a convolutional attention model that produces concise, name-like summaries of source codes. Building on the Transformer architecture, [16] introduces a model that features relative positional encoding and a copying mechanism to address long-range dependencies. Wu et al. [63]devises a structure-induced Transformer that incorporates multi-view graph matrices into its self-attention mechanism. As technological progress persists, numerous studies are now incorporating structural code information to enhance feature representation. Hu et al. [19] propose a Structure-based Traversal method that transforms the ASTs into a sequential format for LSTM training. EASLE leverages a multi-task learning paradigm to train encoders, such as CodeBERT, enhancing the alignment learning between code and summaries, ultimately enhancing the quality of source code summarization. Furthermore, many studys [8, 28] employ Graph Neural Networks (GNNs) to represent the structural aspects of source code. However, they lack the capability to handle large-scale codebases [18, 32].

Recent studies indicate that LLMs outperform smaller models specifically trained for particular natural language processing tasks [22, 52]. As highly parameterized LLMs continue to evolve, there is a discernible shift from encoder-decoder architectures toward decoder-only models, such as Codex, particularly for tasks like code summarization. For example, GitHub Copilot, based on the CodeX model [7], provides real-time coding suggestions and completions, while StarCoder [36] integrates functionalities like code completion, error detection, and code optimization suggestions, significantly aiding developers in understanding and writing complex code. Sun et al. [54] design several heuristic questions/instructions to gather feedback from ChatGPT and evaluate its performance on zero-shot code summarization tasks. This approach helped guide ChatGPT in generating code summaries that align with the desired distribution. Haldar [22] investigates the performance of LLMs of code summarization and argues that the effectiveness of these models depends on the degree of token overlap, specifically subword tokens, between the code and its corresponding natural language descriptions. Fried et al. [14] introduces InCoder, a large language model, and experiment with zero-shot training using the CodeXGLUE Python dataset. Chen et al. [7] fine-tune CodeX for the code summarization task and introduce a novel variant, CodeX-D. However, it is essential to note that excessive data usage in few-shot learning or fine-tuning can lead to catastrophic forgetting within the model [68]. Therefore, we propose the use of a novel technical paradigm involving prompt-based fine-tuning for the task of code summarization.

2.2 Prompt-tuning for LLMs

LLMs accumulate extensive world knowledge, prompting researchers to adapt them to specific downstream tasks with minimal resource use [59]. Task-oriented fine-tuning [30, 65] is proven to bridge the gap between pre-training tasks and real-world applications effectively. However, this approach can constrain the model's ability to generalize across different fields [1].

Prompt-tuning offers a simpler and more efficient alternative by adding a trainable prefix or continuous vector to LLMs with frozen parameters [38, 39, 58, 59]. For instance, methods based on meticulously constructed few-shot sample scenario learning [1, 17, 47] aim to enhance the model's adaptability and potential for specific tasks. Meanwhile, the CoT method [34, 61] progressively

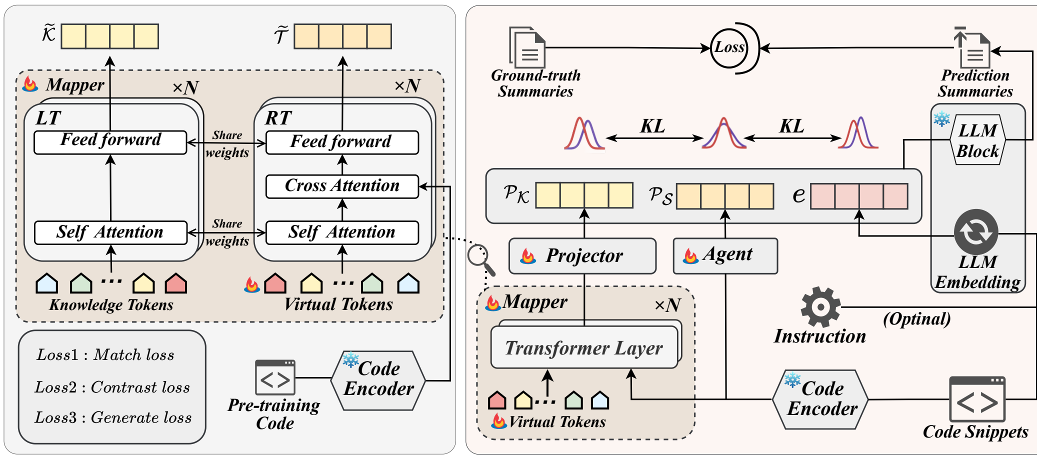


Fig. 2. Overview of EP4CS. **Left:** In the first stage, the *Mapper* connects with frozen *code encoder*, employing $< \mathcal{C}, \mathcal{K} >$ pairs for pre-training. This phase strategically aligns programming languages with natural languages, enhancing their mutual information through three specialized subtasks(detailed in Section 3.3). **Right:** During the second stage, the Pre-trained *Mapper*, after processing through *Projector*, simulates the generation of new code snippets' knowledge prompts $\mathcal{P}_{\mathcal{K}}$,and the *Struct-Agent* extracts structural information from the *Code Encoder* using VAEs and followed by the generation of $\mathcal{P}_{\mathcal{S}}$. $\mathcal{P}_{\mathcal{K}}$ and $\mathcal{P}_{\mathcal{S}}$ are meticulously aligned with the LLM's embedding space via Kullback-Leibler(KL) divergence. Upon receiving instructions, the LLM efficiently generates accurate summaries of the code, utilizing the strategic support of prompt vectors.

guides large models, enabling them to tackle complex problems. These techniques, however, rely on complex text prompt templates that consume significant contextual resources. Finding the optimal template is challenging for both manual and automated methods, which restricts their task processing flexibility [33]. Wang et al. [59] proposes inserting adapter modules into the pre-trained model and only fine-tuning these parameters to adapt to different programming languages, which can significantly overcome catastrophic forgetting in multilingual fine-tuning. The prefix-tuning method [58] delivers impressive results in generation tasks; continuous differentiable virtual tokens are easier to optimize and more effective than traditional discrete text. The P-Tuning [38, 39] method freezes all main model parameters and incorporates a short trainable soft prompt vector at the start of training data for loss optimization. Research [38] indicates that when the parameter size reaches 10 billion, the impact of prompt-tuning is comparable to fine-tuning.

Addressing these challenges is essential to fully realizing the potential of LLMs in software engineering, enabling further advancements in automation, precision, and security. This paper employs prompt-tuning techniques to adapt LLMs for the specific task of code summarization, demonstrating promising results under low-resource conditions.

3 Framework

3.1 Overview

Fig. 2 provides an overview of EP4CS, which comprises a two-stage training process and includes four key components: the *Code Encoder*, *Mapper*, *Struct-Agent*, and *LLM*. EP4CS is designed with dynamically adjustable, task-optimized continuous prompt vectors, significantly enhancing the ability of LLMs to understand and generalize for specific tasks. **To overcome the first challenge**, the *Mapper* merges the benefits of both discrete and continuous prompts. In the face of multiple pre-training task constraints, its internal virtual tokens effectively simulate the background knowledge for the specified code. **In response to the second challenge**, the *Struct-Agent* employs the *Code Encoder* and Variational Autoencoders (VAEs) to create structured prompts that enable the LLM to parse the structural information of the code precisely.

3.2 Notations

First, we define three essential collections that form the foundation of this paper:

- **Code Snippets Collection**: Denoted as $C = \{c_1, c_2, \dots, c_N\}$, where each c_i represents an individual code snippet. This collection serves as the core of our analysis, and N refers to the total number of code snippets in the set.
- **Natural Language Summary Collection**: Denoted as $\mathcal{S} = \{s_1, s_2, \dots, s_N\}$, where each s_i is a natural language description corresponding to c_i. These summaries explain the functionality and purpose of the respective code snippets.
- **Knowledge-enhanced Collection**:Building on prior research [2], the knowledge is represented as $\mathcal{K} = \{k_1, k_2, ..., k_N\}$. Each k_i encapsulates multi-dimensional background information, including data flow dependencies, identifier semantics derived from Treesitter [57] analysis, and contextual details related to third-party library usage. This comprehensive knowledge representation enhances the ability of LLMs to better understand and manage code-related tasks.

A bijective relationship exists among the elements c_i, s_i, and k_i of the respective sets.

3.3 Algin Code Snippets & Knowledge via Mapper

Previous continuous prompting methods overlook the input of specialized background knowledge (e.g., technology stack information). This omission may lead to outputs from LLMs that deviate from the expected goals. Therefore, we present a component called *Mapper*, which uses a variety of pre-training tasks to align code snippets with background knowledge. As shown in Fig. 2(a), *Mapper* consists of two parts: the Left Transformer (LT) encoder and the Right Transformer (RT) encoder. The overall process when processing the pre-trained data pair $< C, \mathcal{K} >$ can be described as follows: First, the frozen *Code Encoder* generates a representation $\tilde{C}$ from the input, and then the $\tilde{C}$ is input to the RT encoder. Meanwhile, the LT encoder processes the background knowledge $\mathcal{K}$, extracting a higher-order representation $\tilde{\mathcal{K}}$. The RT encoder then integrates the virtual tokens $\mathcal{V}_M$ and the code representation $\tilde{C}$, producing a simulated knowledge representation $\tilde{\mathcal{T}}$, designed to mimic human knowledge processing. The steps are formalized as follows:

$$\begin{aligned} \tilde{\mathcal{K}} &= Enc_{RT}(Enc_{LT}(\mathcal{K}; \theta_{\mathcal{F}}); \theta_{\mathcal{F}}) \\ \tilde{\mathcal{T}} &= Enc_{LT}(CrossAttention(\mathcal{V}_M, \tilde{C}); \theta_{\mathcal{F}}, \theta_M), \end{aligned} \tag{1}$$

where $\theta_{\mathcal{F}}$ represents the model parameters shared between LT and RT, θ_M denotes the parameters of the virtual tokens. We will next detail how to design pre-training tasks to enable the *Mapper* to align among code, background knowledge, and virtual tokens.

3.3.1 Contrastive Learning. Contrastive learning enhances the model's ability to extract key features that distinguish between positive and negative samples by explicitly differentiating them. Within the framework of contrastive learning, we optimize the emulation of virtual tokens by maximizing the mutual information between the simulated knowledge representations outputted by the LT encoder and the RT encoder. Initially, we compute pairwise similarities of between $\widetilde{\mathcal{T}} = \{\tilde{t}_1, \tilde{t}_2, ..., \tilde{t}_N\}$ and $\widetilde{\mathcal{K}} = \{\tilde{k}_1, \tilde{k}_2, ..., \tilde{k}_N\}$. Then, we select the sample pairs with the highest similarity scores as positive samples. Concurrently, other pairs with lower similarity or irrelevant features are considered negative samples. During the optimization process, we adjust model parameters using the InfoNCE loss function, which is defined as:

$$\mathcal{L}_{\text{contra}} = -\log \frac{\exp\left(\text{sim}(\tilde{t}_i, \tilde{k}_i)/\tau\right)}{\sum_{i \neq j} \exp\left(\text{sim}(\tilde{t}_i, \tilde{k}_j)/\tau\right)}, \tag{2}$$

where τ is the temperature parameter and $sim(*)$ denotes the similarity function. We reference the approach in [20] and use a unimodal attention mask. This ensures that during the attention calculation, positive and negative samples do not share information.

3.3.2 Knowledge-grounded Text Generation. We design a generative loss to further enhance the modeling capabilities of the *Mapper*. Its training objective is to train the *Mapper* to produce outputs that align with the knowledge text $\widetilde{\mathcal{K}}$ using the given code representation $\widetilde{C}$. In the *Mapper*'s design, the code feature $\widetilde{C}$ first passes through the self-attention layer, then is transferred to the *RT* Encoder via shared parameters, allowing V_M to capture more code feature information. Drawing on the [35, 67], we use a mask to control the interaction between *LT* and *RT*, and use the [DEC] token as a signal to initiate decoding in LT. Let $\widehat{\mathcal{K}} = \{\widehat{k}_1, \widehat{k}_2, \ldots, \widehat{k}_N\}$ be the text sequence that *Mapper* is to generate, finally, the *Mapper* is optimized through the cross-entropy loss:

$$\mathcal{L}_{gen} = -\sum_{i=1}^{N} k_i \log \frac{\exp(\widehat{k}_i)}{\sum_{j=1}^{N} \exp(\widehat{k}_j)}, \tag{3}$$

where N represents the number of samples in the collection.

3.3.3 Code-Knowledge Matching Loss . To enhance the effectiveness of the *Mapper* module, it's essential to precisely align code representation and *RT*'s output. A key challenge in this context is the development of robust and impactful negative samples. We postulate that a closer alignment between the computational representations of code and the corresponding background knowledge ($< C, \mathcal{K} >$ pairs) can significantly enhance *Mapper*'s capability. Building on the research [49, 66], we employ a bidirectional matching loss. This loss can address the complex interactions between code and knowledge, advancing our comprehension of complex linguistic structures by ensuring that the loss function effectively captures the essence of both entities. The loss function is as follows:

$$\mathcal{L}_{match} = \sum_{(\tilde{c}_i, \tilde{t}_i)} \left(\begin{array}{c} \left[\gamma - s(\tilde{k}_i, \tilde{c}_i) + s(\tilde{k}_i, \tilde{c}_j)\right]_+ \\ + \left[\gamma - s(\tilde{c}_i, \tilde{k}_i) + s(\tilde{c}_i, \tilde{k}_j)\right]_+ \end{array} \right), \tag{4}$$

where γ represents the margin hyperparameter, which defines the threshold for distinguishing between positive and negative samples. The $[x]_+ \equiv \max(x, 0)$ defines the rectifier operation and $s(*)$ denotes the distance function, respectively. Together, these operations significantly improve the model's generalization ability and robustness, enabling it to perform well across diverse scenarios.

3.4 Generative Learning from Frozen LLM

3.4.1 Struct-Agent. Although LLMs are widely considered to excel in understanding and generalization abilities, their training mode based on decoding does not adequately consider the structural semantics of code. To address this issue, we design a *Struct-Agent* component based on variational autoencoders, which models the structured representation of code as a probability distribution through variational inference. Specifically, the *Struct-Agent* obtains vectors from a pre-trained code language model and then generates latent variables z_c through the agent's encoder. Then, the decoder reconstructs the original code from the latent space of z_c to ensure that the reconstructed code retains as much of the original structural and semantic features as possible. Below are the specific implementation steps:

$$
\begin{aligned}
q_\phi(z_c|\widetilde{C}) &= \mathcal{N}\left(z_c|\mu_\phi(\widetilde{C}), \mathrm{diag}\left(\sigma^2_\phi(\widetilde{C})\right)\right) \\
p_\varphi(\widetilde{C}|z_c) &= \mathcal{N}\left(\widetilde{C}|\mu_\varphi(z_c), \mathrm{diag}(\sigma^2_\varphi(z_c))\right),
\end{aligned} \tag{5}
$$

$$
\mathcal{L}_{agent} = \mathbb{E}_{q_\phi(z_c|\widetilde{C})}[\log p_\varphi(\widetilde{C}|z_c)] - D_{KL}(q_\phi(z_c|\widetilde{C})\|p(z_c)), \tag{6}
$$

where μ and σ represent the mean and variance of vector z_c, respectively, and ϕ and φ represent the parameters of the encoder and decoder in the *Struct-Agent*. Compared to past methods, the agent enables the latent variable z_c to be trained as a potential structured feature.

3.4.2 Fusion Embedding Generation. During training and inference, the code snippet c is input into the LLM to obtain code embeddings. Additionally, the *Mapper* and the *Struct-Agent* output vectors z_c and $\widetilde{\mathcal{T}}$, respectively. These outputs are then fed into the projector, which ensures the output dimensions are consistent with the embedding dimensions of the LLMs, to generate the information-rich continuous prompt vectors:

$$
\mathcal{P}_\mathcal{K} = MLP(\tilde{\mathcal{T}}), \quad \mathcal{P}_S = MLP(z_c). \tag{7}
$$

Similar to traditional prompting methods that integrate discrete prompts with code snippets, EP4CS also concatenates $\mathcal{P}_\mathcal{K}, \mathcal{P}_\mathcal{S}$ with the embedding vector e. Prompt vectors are strategically placed at the beginning, and code vectors are positioned at the end to optimize the flow of information. Unlike previous methods, we introduce a regularization mechanism to mitigate the impact of distributional discrepancies by adjusting the weighting of the prompt and code vectors, ensuring that the model remains effective across varying data distributions. The regularized prompt vectors are then used to guide the LLM in autoregressive output. The implementation steps are as follows:

$$
\mathcal{L}_{KL} = \mathcal{D}_{KL}\left(\mathrm{concat}(\mathcal{P}_\mathcal{K}, \mathcal{P}_\mathcal{S}) \parallel LLM(c)\right), \tag{8}
$$

where $\mathcal{D}_{KL}$ represents the KL divergence, which is used to measure the difference between and the target distribution.

3.4.3 Joint Optimization. In our framework, we implement a two-stage independent training strategy. In the first stage, the total loss is calculated as the weighted sum of $\mathcal{L}_{contra}$, $\mathcal{L}_{gen}$, and $\mathcal{L}_{match}$. In the second stage, we freeze all parameters of the LLM and focus on optimizing the *Mapper*, *Projector*, and the *Struct-Agent*. The optimization goal for this stage is to minimize the difference between the generated code summaries and the ground-truth summaries. The joint optimization loss function is modeled as follows:

$$
\mathcal{L}_{stag2} = -\lambda_1 \sum_{i=1}^{C} s_i \log \frac{\exp(\hat{s}_i)}{\sum_{j=1}^{C} \exp(s_j)} + \lambda_2 \mathcal{L}_{KL} + \lambda_3 \mathcal{L}_{agent}, \tag{9}
$$

where $\hat{s}_i$ represents the probability vector corresponding to the predicted summary, and y_i is the ground-truth summary. λ_1, λ_2, and λ_3 represent the weights of the respective losses, used to balance the impact of each loss term on model training. These weights are set to 1.0, 0.5, and 0.1 based on empirical adjustments and experimental settings.

4 Evaluation and Analysis

In this section, we introduce the experimental setup, which includes the dataset, evaluation metrics, compared baselines, and the hyperparameter configuration of EP4CS. We also conduct a series of experiments to answer the following research questions (RQs):

- **RQ1:** How does EP4CS perform relative to other methods of code summarization?
- **RQ2:** Is EP4CS adaptable to different types of large language models?
- **RQ3:** How do structured information and background knowledge impact the performance of large language models?
- **RQ4:** What impact do the sizes of virtual tokens and structured prompts have on the generation of code summaries by large language models?
- **RQ5:** What is the resource consumption of EP4CS during its training phase?
- **RQ6:** What is the performance of EP4CS in human evaluations?

4.1 Dataset

The CodeSearchNet dataset [26], is extensively employed for code summarization and other tasks. Offering a diverse collection of code samples, it supports the development of models capable of summarizing code across different programming languages and domains. However, the original dataset contains considerable noise. To address this, we adopt a refined framework from the CodeXGLUE code-to-text dataset [40], which filters out unparseable entries, non-English documentation, excessively short or long documentation, and entries with special markers. Latee, as outlined in section 3. 2, we construct $< \mathcal{C}, \mathcal{S}, \mathcal{K} >$ tuples and use these dataset splits for our experiments. To facilitate comparison with existing methods and baselines, we have chosen Python and Java as the languages for comparison.

4.2 Evaluation Metrics

To evaluate the performance of EP4CS in code summarization, we utilize a comprehensive set of widely recognized evaluation metrics: BLEU [45], ROUGE-L, METEOR [5], and SentenceBERT [50]. These metrics are critical in providing a well-rounded assessment, capturing both syntactic and semantic qualities of the generated summaries.

- **BLEU** is extensively used in text generation tasks within natural language processing. It measures the similarity of generated code summaries to actual code summaries by calculating the precision of n-grams and penalizes for insufficient summary lengths.
- **ROUGE-L** is a variant of the ROUGE metric, which calculates similarity using the Longest Common Subsequence (LCS). It assesses similarity in code summarization tasks by comparing the LCS of the generated and actual summaries.
- **METEOR** is a comprehensive metric that evaluates summaries based on word-level match, considering recall, precision, and syntactic fluency.
- **SentenceBERT**. Unlike the three metrics mentioned above that primarily assess textual similarity between the ground truth and generated summaries, SentenceBERT evaluates semantic similarity. It transforms the compared summaries into embeddings in a unified vector space and then measures their semantic similarity using cosine similarity.

4.3 Base Methods & Baseline Methods

4.3.1 *Code Pre-trained Language Models.*

- **CodeBERT [13]:** CodeBERT uses masked language modeling and token replacement detection to handle coding tasks and their descriptions effectively. It is particularly skilled at code search and documentation writing due to its deep understanding of code's syntax and annotations.

Table 1. The large language models employed in this paper and its particulars.

Framework	Hidden Size	Vocabulary Size	Max Sequence Length	Code Fine-tuning †
Qwen1.5$_{0.5B}$	1,024	151,936	32,768	No
Qwen1.5$_{4B}$	2,560	151,936	32,768	No
Qwen1.5$_{7B}$	4,096	151,936	32,768	No
PolyCoder$_{160M}$	768	50,304	2,048	Yes
PolyCoder$_{0.4B}$	1,024	50,304	2,048	Yes
PolyCoder$_{2.7B}$	2,560	50,304	2,048	Yes
StarcoderBase$_{1B}$	2,048	49,152	8,192	Yes
StarcoderBase$_{3B}$	2,816	49,152	8,192	Yes
StarcoderBase$_{7B}$	4,096	49,152	8,192	Yes

†Whether the model has been fine-tuned using a code-related corpus.

- **UniXCoder [20]:** UniXCoder combines the pre-training tasks of code understanding and generation. It integrates cross-modal content like Abstract Syntax Trees (AST), which enhances the model's ability to express code.
- **CodeT5 [60]:** CodeT5 is engineered for a diverse range of programming language tasks, leveraging the T5 architecture to facilitate summarization, completion, translation, and documentation.

4.3.2 Large Language Models.

- **PolyCoder [23]:** PolyCoder is released by researchers from Carnegie Mellon University. It is trained using the GPT NeoX toolkit on 249GB of code from 12 programming languages, and is available in three sizes: 160M, 0.4B, and 2.7B.
- **Qwen1.5 [4]:** Qwen is part of Alibaba's Tongyi Qianwen series, showing high performance across various benchmarks. It handles complex language understanding and generation tasks effectively. Versions are available with 0.5B, 4B, and 7B parameters.
- **StarCoderBase [37]:** Jointly launched by Hugging Face and ServiceNow in 2023, the model has been trained on over 80 programming languages. This training has rendered it an exceptionally diverse model within the programming language domain.
- **ChatGPT [44]:** In order to better compare with other work, we chose CodeX model code-davinci-002. Even though it's not the most advanced model, it still does a great job at code generation, and we set the temperature to the default value of 0.5 to get a clear answer from CodeX.

4.3.3 Baselines.

- **ASAP [2]:** ASAP enhances code context comprehension by incorporating three types of semantic features: repository names and paths, annotated identifier information, and data flow graphs. Initially, BM25 retrieves relevant examples from the sample pool. Subsequently, semantic features are extracted through static analysis and embedded into the few-shot prompt structure, enriching the model's contextual understanding.
- **MICG [17]:** MICG introduces an approach that leverages the in-context learning paradigm by providing large language models with appropriate prompts, such as ten or more examples, to significantly enhance performance in generating code comments encompassing multiple intents.
- **PromptCS [53]:** PromptCS employs deep learning prompt encoders to transform learnable tokens into continuous vector forms. This process has been refined through supervised training, enhancing its suitability for large language models. This method significantly reduces the demand for training resources compared to traditional, manually written discrete prompts.

Table 2. Experimental results of code summary. We report the mean accuracy with a standard deviation of 5 runs. Highlighted are the top performance, while underlined parts are the second best. $\mathcal{B}$: BLEU; $\mathcal{M}$: METEOR; $\mathcal{R}$: ROUGE-L; $\mathcal{S}$: SentenceBERT.

Methods	LLM	Python				Java			
		$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$
ASAP	CodeX_{175B}	17.31	12.11	<u>38.62</u>	56.34	17.67	12.01	34.52	57.88
ASAP	$\text{StarCoderBase}_{1B}$	7.84	3.61	13.42	15.58	8.34	7.46	19.65	30.14
MICG	CodeX_{175B}	18.76	<u>13.43</u>	36.92	54.26	18.64	13.94	36.21	57.94
MICG	$\text{StarCoderBase}_{1B}$	12.42	8.82	23.16	34.80	13.65	8.95	25.03	47.91
PromptCS	$\text{StarCoderBase}_{1B}$	13.88	9.20	27.85	41.03	20.50	14.05	39.47	61.68
PromptCS	$\text{StarCoderBase}_{3B}$	<u>19.75</u>	13.10	38.14	<u>58.15</u>	<u>20.87</u>	<u>14.50</u>	<u>40.23</u>	<u>62.39</u>
EP4CS	$\text{CodeBERT+StarCoderBase}_{1B}$	18.33	12.97	38.17	57.93	18.89	13.12	36.88	58.10
EP4CS	$\text{CodeT5+StarCoderBase}_{1B}$	19.12	13.74	38.21	58.46	21.25	14.83	40.89	61.52
EP4CS	$\text{UniXcoder+StarCoderBase}_{1B}$	**19.81**	**13.74**	**38.40**	**58.66**	**21.85**	**15.04**	**41.22**	**62.67**

4.4 Experimental Settings

For the frozen code encoder, we have selected UniXcoder as the backbone model due to its strong benchmark performance. We refrain from fine-tuning these frameworks on specific datasets in order to preserve their broad knowledge of programming languages, rather than adapting them to the particularities of individual datasets. The *Mapper* consists of 12 layers of Transformer blocks, each with 12 attention heads. We set the code truncation length to 256, the mini-batch size to 16, and the learning rate to 5e-5, with a plan for 100,000 pre-training steps.

During the second training stage, we implement a linear learning rate decay strategy that reduces the learning rate from 5e-5 to 1e-6 over a 2,000-step warm-up period. We adjust the mini-batch size to 4 and use the AdamW optimizer. To enhance the training process and prevent overfitting, we present an early stopping mechanism based on the BLEU score of the validation set, with an early stop factor of 2 and a maximum of 10 epochs. We extend the input sequences to their maximum length using special tokens from the LLM's vocabulary. Both training stages and the baseline are developed using the PyTorch 2.1.0 framework and Python 3.9. All experiments were conducted on a server equipped with an 80 GB Nvidia A100 GPU, running Ubuntu 20.04.

5 Results

5.1 RQ1: Compare with Other Benchmark Experiments

To answer RQ1, we evaluate different benchmarks, and the results are presented in Table 2:

First, as shown in the first and third rows of Table 2, the ASAP and MICG methods deliver strong performance when applied to CodeX. For instance, they achieve BLEU scores of 17.31 and 18.76, respectively, on the Python dataset, and SentenceBERT semantic scores of 56.34 and 54.26. These results demonstrate the practical effectiveness of discrete prompt learning approaches for code summarization. However, when the parameter count of the backbone model is reduced to 1B, both methods experience a substantial decline in performance. This suggests that current discrete prompting strategies may not fully leverage the capabilities of large language models. Future work should therefore explore more expressive or adaptive prompting techniques to bridge this gap.

In contrast, PromptCS—the method leveraging continuous prompts—delivered strong performance on the Java dataset, achieving metric scores of 20.87, 14.50, 40.23, and 62.39. Although PromptCS performs well across multiple tasks, its performance on the Python dataset is relatively weaker. For instance, when using StarCoder 1B, the four evaluation metrics for Python were only 13.88, 9.20, 27.85, and 23.80, respectively. This phenomenon is primarily attributed to the inherent flexibility of the Python language—particularly in terms of variable naming and code structure. Java's stricter coding standards, by comparison, enable higher consistency between generated and reference summaries.

Most notably, EP4CS demonstrated consistent and significant performance gains across both Java and Python datasets, regardless of the underlying language model. When paired with the same StarcoderBase$_{1B}$ backbone used in PromptCS, EP4CS improved performance by up to 8.24%, 7.05%, 4.43%, and 1.61% on Java. On the python language, it achieved scores of 19.81, 13.73, 38.40, and 58.66—highlighting its robustness across languages. These results affirm the effectiveness of incorporating contextual knowledge and code structure into continuous prompts, significantly enhancing the alignment between model-generated and reference summaries.

Furthermore, our experiments show that the choice of CodePLM has a marked impact on model performance. On both datasets, integrating UniXcoder as the core model within EP4CS yielded improvements of 1.2% and 7.8% on SentenceBERT compared to using CodeBERT. This underscores the strategic importance of selecting specialized codePLMs to enrich LLMs with domain-specific knowledge and optimize performance on code understanding tasks.

Key Findings

EP4CS outperforms existing baseline models across all evaluation metrics, demonstrating its tremendous potential in code summarization tasks. It overcomes the shortcomings of previous prompt-based methods and excels in terms of robustness.

Table 3. Effectiveness of EP4CS on the CSN-Python dataset. Compare zero-shot learning (without task-specific examples) and few-shot learning (with a small number of examples). $\mathcal{B}$: BLEU; $\mathcal{M}$: METEOR; $\mathcal{R}$: ROUGE-L; $\mathcal{S}$: SentenceBERT.

LLM	Parameter Scale	Zero-shot				Few-shot				EP4CS			
		$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$
PolyCoder	160M	8.95	3.80	14.82	13.10	8.39	3.36	12.23	19.09	**13.72**	**11.39**	**28.81**	**49.31**
	0.4B	9.56	3.13	15.49	10.01	8.24	4.03	13.37	19.59	**14.02**	**12.69**	**30.48**	**53.12**
	2.7B	9.80	2.76	15.83	8.74	9.68	4.44	15.91	21.62	**15.29**	**12.93**	**32.28**	**54.08**
Qwen1.5	0.5B	7.76	12.43	21.11	44.89	8.73	7.39	15.18	37.76	**16.96**	**12.13**	**35.38**	**55.08**
	4B	7.89	12.17	19.31	46.18	9.79	4.22	15.43	21.49	**17.63**	**12.06**	**34.51**	**57.25**
	7B	7.14	12.06	17.97	47.28	10.80	7.60	19.46	30.28	**19.73**	**14.42**	**39.25**	**59.79**
StarCoderBase	1B	8.42	4.81	13.50	16.32	10.06	6.74	17.82	27.69	**19.81**	**13.74**	**38.40**	**58.66**
	3B	9.81	4.23	17.23	13.27	12.67	8.52	23.48	35.35	**20.51**	**14.22**	**39.35**	**59.63**
	7B	10.54	8.50	21.40	27.77	12.36	8.46	22.88	34.27	**20.42**	**14.39**	**39.66**	**59.64**
ChatGPT	-	9.41	14.01	20.32	52.89	14.21	15.27	33.87	58.11	-†	-†	-†	-†

†ChatGPT is a non-open source model and we cannot train it.

5.2 RQ2: Comparative Performance of EP4CS across Different Large Language Models

To investigate the performance of our framework when combined with other large language models, we set up the following configurations :

(1) **Zero-shot:** Instruct the LLM to summarize code snippets using only manual prompts, a typical prompt might be: //**Human**: You are a helpful code summarizer. Please describe in simple English the purpose of the following Java code snippet: <*code*> //**Assistant**:
(2) **Few-shot:** Concatenate M code-summary pairs without providing any additional instructions. Each pair, consisting of a code snippet and its corresponding summary, is randomly selected from the candidate pool: $<code_1\ ,\ \ summary_1>,\ \ \ldots,\ \ <code_M\ ,\ \ summary_M>$;
(3) **EP4CS**: Building on the findings from RQ1, we adopt UniXcoder as the backbone model. To ensure a fair comparison across experimental setups, we employ prompt instructions aligned with the zero-shot paradigm.

Firstly, as shown in Table 3, the results clearly demonstrate a trend in the code summarization task: model performance improves as the number of parameters increases. Specifically, using the PolyCoder model at varying scales—160M, 0.4B, and 2.7B—yielded BLEU scores of 13.72, 14.02, and 15.29, respectively. These results indicate a positive correlation between parameter scale and model performance, suggesting that larger models are better equipped to capture complex patterns. Moreover, in most cases, LLMs do not perform well in a zero-shot setting, indicating that their outputs are not drawn from the models' inherent knowledge base. This limitation may be due to the failure to activate domain-specific knowledge during inference.

Secondly, our continuous prompting strategy significantly outperforms two conventional discrete prompting frameworks across multiple models and scales. Notably, on the SentenceBERT metric, EP4CS consistently achieves performance improvements that are often double those of the competing methods. This remarkable enhancement is likely due to EP4CS's ability to explore a broader solution space, enabling it to identify optimal solutions that discrete methods may overlook.

Furthermore, although Qwen1.5 has a comparable or even larger number of parameters, it generally under performs compared to both StarCoderBase and PolyCoder—the latter having significantly fewer parameters. This discrepancy in performance may be attributed to Qwen1.5's lack of fine-tuning on code-specific datasets. In contrast, Qwen1.5 tends to produce more general natural language output, which likely contributes to its unusually high SenTenceBERT score. StarCoderBase, on the other hand, has been explicitly pretrained on code corpora encompassing over 80 programming languages. Consequently, its embedding space is more closely aligned with the semantic characteristics of code, resulting in superior performance on software-related tasks.

Finally, synthesizing results from Table 2 and Table 3, we observe that employing larger and more robust codePLMs and LLMs as backbone models markedly boosts EP4CS's performance. This trend underscores the decoupling characteristics of our framework, reinforcing the notion that increasing parameter scale positively impacts performance. However, we can also observe that the growth of model parameters and performance is not linear. For example, when StarCoderBase is expanded from 1B to 3B, the BLEU score increases by 3.53% , but when further expanded to 7B, the performance tends to plateau, with a score of 20.42.

Key Findings

EP4CS consistently outperforms zero-shot and few-shot across LLM scales (from 160M to 7B parameters). Starcoderbase benefits more from EP4CS, but its performance stabilizes beyond 3B parameters, indicating diminishing returns for very large models.

5.3 RQ3: Blation Study on Each Component in EP4CS

This study shows that knowledge-enhanced, structured prompt vectors significantly improve the performance of large language models. We chose StarCoderBase$_{3B}$ as the base backbone model

Table 4. Performance of EP4CS on Java and Python Datasets After Removing Various Components. Based on the StarCoderBase$_{3B}$ Backbone Model.

Components	Python				Java			
	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$	$\mathcal{B}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{S}$
random	6.95	4.45	10.25	27.72	4.67	4.03	10.22	23.42
w/o $\mathcal{P}_\mathcal{S}$&$\mathcal{P}_\mathcal{K}$	9.64	8.64	20.70	33.93	5.95	4.27	11.52	24.68
w/o $\mathcal{P}_\mathcal{K}$	18.01	13.69	35.88	55.71	18.58	13.38	38.76	57.61
w/o $\mathcal{P}_\mathcal{S}$	19.03	14.59	37.26	57.68	20.03	14.06	39.51	59.29
EP4CS	**20.51**	**14.22**	**39.35**	**59.63**	**21.67**	**14.79**	**39.67**	**61.85**

to analyze the specific impact on overall model performance when different key components are stripped away. As shown in Table 4, "*w/o* $\mathcal{P}_\mathcal{S}$" represents the removal of the structural prompt vector, i.e., the Agent module; "*w/o* $\mathcal{P}_\mathcal{K}$" indicates the cancellation of the *Mapper*'s output, i.e., not involving the knowledge vector; and "*w/o* $\mathcal{P}_\mathcal{K}$ & $\mathcal{P}_\mathcal{K}$" means both of the aforementioned components are removed, employing a zero-shot. Moreover, the "*random*" method retains both types of prompt vectors, but these vectors are initialized randomly.

The analysis results reveal that removing the structural cue vectors led to a 7.22% and 7.57% drop in BLEU scores, and a 3.27% and 4.14% decrease in SentenceBERT scores, respectively. Eliminating the knowledge cue vectors resulted in reductions of 12.19% and 14.26% in BLEU scores, and 6.57% and 6.86% in SentenceBERT scores. This suggests that removing one type of cue vector has a more minor performance impact than removing both, indicating an overlap in their functions. Furthermore, assigning values to these vectors through random initialization proved less effective than removing them altogether. The random vectors, with their non-specific configurations, significantly hindered the model's understanding. This analysis underscores the significance of our strategic framework in enhancing model performance.

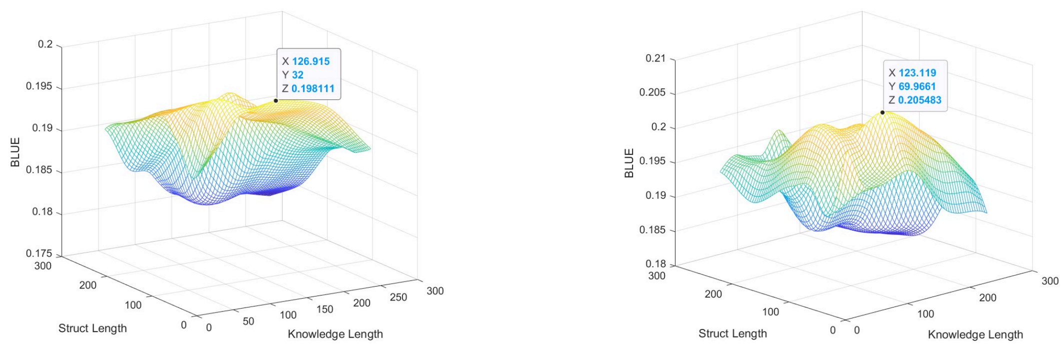
Key Findings

EP4CS's dual-prompt architecture provides synergistic benefits. Additionally, randomly initializing the prompt vectors leads to worse performance than removing them altogether.

5.4 RQ4: Influence of Key Configurations on EP4CS

Table 5. The table displays the performance results based on the BLUE metric for various token sizes. The columns represent structure prompts, while the rows correspond to background prompts.

Size	StarCoderBase$_{1B}$						StarCoderBase$_{3B}$					
	32	**64**	**96**	**128**	**192**	**256**	**32**	**64**	**96**	**128**	**192**	**256**
32	19.18	19.58	19.74	**19.81**	19.66	19.05	20.31	19.68	19.70	20.32	19.05	18.86
64	18.96	19.80	19.67	19.72	19.45	18.93	19.27	19.64	19.87	**20.59**	20.05	19.31
96	19.32	19.72	19.64	19.43	19.23	18.92	19.61	20.20	20.23	20.29	19.91	18.97
128	19.54	19.59	19.62	19.65	18.92	18.78	19.54	20.30	20.17	20.14	19.86	18.46
192	19.31	19.34	19.06	18.29	18.26	18.23	19.13	19.70	19.05	19.68	18.40	18.13
256	19.08	18.56	18.44	18.16	18.09	17.96	19.41	19.58	19.49	19.59	18.36	18.02



(a) Based On StarCoderBase$_{1B}$ (b) Based On StarCoderBase$_{3B}$

Fig. 3. The figure presents the performance results for various sizes of structured and knowledge prompts. We selected the Python dataset and utilized UniXcoder as the foundational model. The figure emphasizes the performance achieved with the optimal configuration.

As we explore innovative methods in our research, EP4CS relies heavily on two key configurations that significantly affect model performance: the size of the knowledge prompt and the size of the structured prompt. These two components—represented by virtual tokens and generated by the *Mapper* and *Struct-Agent*, respectively—play a critical role in determining the model's ability to retain and process information.

To systematically determine the optimal sizes for these vectors, we conduct comprehensive tests using various configurations—32, 64, 96, 128, 192, and 256. Our findings, vividly depict in Fig. 3 and Table 5, show that smaller sizes, such as 32, generally provide insufficient context, severely limiting model efficacy. This is also the deficiency of previous continuous prompt methods. Conversely, excessively large sizes, like 256, introduce excessive semantic noise, which paradoxically degrades performance by cluttering the model's output with irrelevant information. Fig. 3 demonstrates a synergistic effect between the sizes of the knowledge and structured prompt vectors. Optimal performance on StarCoderBase$_{1B}$ is achieved with a knowledge prompt size of 128 and a structured prompt size of 32. However, at the 3B scale, the structured prompt size peaks at 64. This configuration strikes an optimal balance between the adequacy of information and processing efficiency, thus enhancing the model's adaptability and accuracy.

Key Findings

The optimal prompt size is model-dependent. Large prompts introduce noise, while small prompts lack context. This highlights the need to tailor prompt design to model scale and task demands for optimal performance.

5.5 RQ5: Evaluating Efficiency and Effectiveness

Firstly, we compare several prompt learning frameworks with task-oriented fine-tuning methods. As shown in Fig. 4, prompt-based methods demonstrate a significant improvement over Zero-Shot, and EP4CS surpasses all prompt methods in both BLEU-4 and SentenceBERT scores. However, we still observe that task-oriented fine-tuning methods provide more stable performance. Nonetheless, EP4CS, through its innovative prompt learning method, can approach or even surpass traditional task-oriented fine-tuning methods in most cases. On PolyCoder, EP4CS achieves BLEU-4 scores comparable to fine-tuning, while its SentenceBERT score significantly outperforms traditional

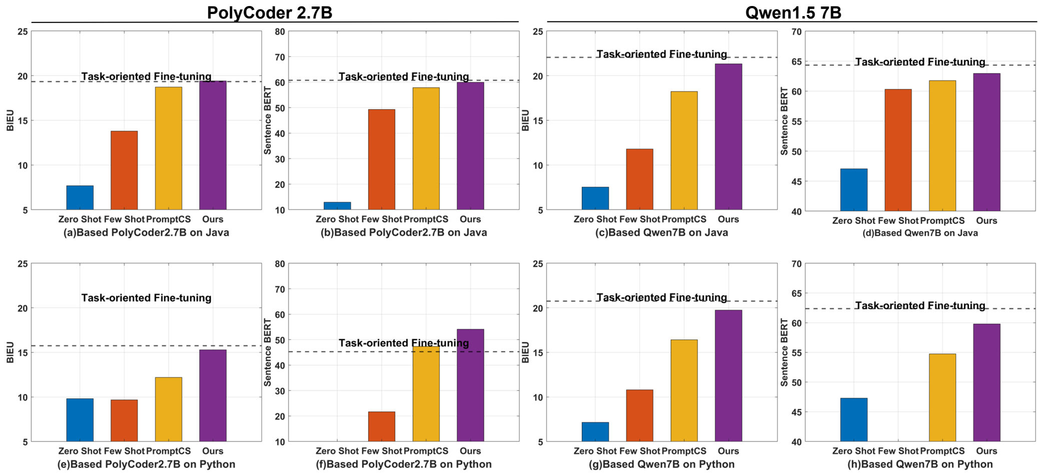


Fig. 4. Relative BLEU-4 and SentenceBERT improvement of EP4CS over others on Qwen1.5$_{7B}$ and PolyCoder$_{2.7B}$. The dashed line indicates the metrics for the task-oriented fine-tuning method. The blank spaces in the chart represent that the model's performance did not meet the minimum value depicted.

Table 6. A comparison of training costs across Task-oriented Fine-tuning, PromptCS, and EP4CS for models of varying scales, shedding light on their efficiency and cost-effectiveness.

LLMs	Parameter Scale	Task-oriented Fine-tuning	PromptCS	EP4CS
PolyCoder	160M	12h:25m	09h:41m	06h:01m
	0.4B	22h:18m	14h:32m	12h:43m
	2.7B	81h:54m	27h:41m	21h:01m
Qwen1.5	0.5B	14h:54m	11h:35m	10h:29m
	4B	45h:18m†	41h:52m	32h:21m
	7B	191h:34m†	77h:41m	58h:43m
StarCoderBase	1B	50h:06m	29h:17m	15h:43m
	3B	82h:26m	43h:51m	23h:43m
	7B	211h:05m†	67h:21m	54h:43m

†is the single-epoch training time for it

fine-tuning, indicating that prompt learning offers a stronger parameter efficiency advantage for medium and small models. On the Qwen 7B model, EP4CS also performs remarkably, especially on the Java task, where the BLEU score reaches 21.32, close to the highest fine-tuning score of 22.05.

In summary of the above RQs, we also observe an interesting phenomenon: semantic-form decoupling. In RQ2, the Qwen1.5, which was not trained on programming language, achieves a very high SentenceBERT score. Meanwhile, EP4CS's SentenceBERT score of 54.08 significantly surpasses fine-tuning's 45.30, but its BLEU is slightly lower than fine-tuning. This suggests that traditional fine-tuning may over-optimize surface form features, while prompt-based methods better preserve deep semantic coherence, which is of crucial value for code generation tasks that require logical consistency.

Additionally, Table 6 presents a comparison of the above-mentioned methods in terms of training cost. While task-specific fine-tuning offers significant advantages in improving the model's performance on specific tasks, its higher resource consumption cannot be overlooked. Without the use of quantization and other framework techniques, one epoch of StarCoderBase$_{7B}$ requires approximately 210 compute hours. In contrast, PromptCS significantly reduces the training time across all large language models, mainly due to its non-intrusive nature, meaning that it does not require updating the parameters of the LLM during training. EP4CS further reduces resource consumption compared to PromptCS, not only because it adopts a continuous prompt fine-tuning strategy, but also because, by introducing the code pre-trained model, EP4CS integrates rich knowledge and programming language features, effectively accelerating the model's convergence process, thereby significantly reducing the difficulty and cost of training.

Key Findings

EP4CS reduces training costs and accelerates convergence through non-invasive prompts and code pre-training. At the same time, it approaches or surpasses task-oriented fine-tuning in performance, and has higher training efficiency.

5.6 RQ6: Human Evaluation

Table 7. Comparison of different frameworks on naturalness, adequacy, and usefulness, Generate summaries based on the StarCoderBase$_{7B}$.

Methods	Naturalness	Adequacy	Usefulness
ASAP	4.0 ± 0.7	3.7 ± 0.7	3.0 ± 1.1
MICG	4.2 ± 0.7	3.6 ± 1.1	3.5 ± 0.8
PromptCS	4.2 ± 0.8	3.9 ± 0.8	3.6 ± 1.1
EP4CS	**4.3 ± 0.7**	**4.1 ± 0.9**	**3.7 ± 1.2**
Original	3.9 ± 0.8	3.7 ± 0.8	3.9 ± 1.3

Although metrics like BLEU, ROUGE-L, and METEOR can assess the lexical similarity between generated summaries and standard answers, they do not fully capture semantic differences. To provide a more comprehensive evaluation of the quality of summaries generated by different methods, this study invited 15 participants: 4 PhD students, 6 Master's students, and five industry developers, all of whom have at least three years of software development experience. We randomly selected 100 code snippets from the test set (50 from Java and 50 from Python). The ratings were based on a 5-point Likert scale. To ensure impartiality, the evaluators were unaware of the summarizations' origins. Additionally, we provided specialized training on summary evaluation for the evaluators. Evaluators were asked to rate the based on three criteria:

(1) Naturalness, assessing the grammatical fluency of the summaries;
(2) Sufficiency, assessing the amount of information in the summaries;
(3) Usefulness, measuring how helpful the summaries are to developers.

Table 7 demonstrates that EP4CS outperforms all others tested, achieving impressive scores of 4.3, 4.1, and 3.7 across the evaluated metrics. Notably, in the domain of Naturalness, every framework achieved scores exceeding 4.0, confirming the proficiency of LLMs in crafting fluent and convincing language. EP4CS stands out uniquely in its ability to enhance Adequacy, as it is the only

```
def make_ar_transition_matrix(coefficients):
    top_row = tf.expand_dims(coefficients, -2)
    coef_shape = dist_util.prefer_static_shape(coefficients)
    batch_shape, order = coef_shape[:-1], coef_shape[-1]
    remaining_rows = tf.concat([
        tf.eye(order - 1, dtype=coefficients.dtype,  batch_shape=batch_shape),
        tf.zeros(tf.concat([batch_shape, (order - 1, 1)], axis=0),
        dtype=coefficients.dtype)
    ], axis=-1)
    ar_matrix = tf.concat([top_row, remaining_rows], axis=-2)
    return ar_matrix
```

(a) A code snippet from the test set

```
#Ground-truth:Build transition matrix for an autoregressive StateSpaceModel .
#Zero-shot:The code snippet is used to make a transition matrix for the AR model.
#Few-shot:Returns a transition matrix from a set of coefficients.
#EP4CS:Creates a matrix that associates the coefficients for an AR model .
```

(b) Summaries generated by different method

Fig. 5. Code summarization case from CSN-Python No. 833

framework to surpass the 4.0 threshold. This distinct strength underscores its superior capability in meeting comprehensive content requirements. However, the challenge remains in the Usefulness metric, where all frameworks fell short of the 4.0 mark, highlighting a prevalent area for future enhancements in LLMs' ability to generate practical code summaries. In addition, we also scored the original data in the dataset. The scores of the three dimensions of the reference summary are concentrated around 3.8, indicating that the quality of the reference summary is relatively low.

6 Discussion

6.1 Case Studies

Fig. 5 illustrates a case of code summarization. In Fig. 5(b), the first line displays the ground-truth summary. Lines 2 to 4 show the summaries generated by StarCoderBase$_{3B}$ using zero-shot, few-shot, and EP4CS approaches for the given code snippet.

In the first case, the ground-truth summary of the function is "Build transition matrix for an autoregressive StateSpaceModel". This can be broken down into two main semantic parts: the first part, "Build transition matrix" (highlighted in blue in Fig. 5(a)), and the second part, "for an autoregressive StateSpaceModel". The code does not explicitly reference StateSpaceModel, and the term "autoregressive" is abbreviated as "ar". Therefore, generating this part of the summary poses a significant challenge for LLMs.

Compared to the ground-truth summary, the analysis is as follows:

1. The zero-shot captures both semantic parts of the actual summary. Although the second part is only presented as "the AR model", considering the difficulty of extracting this content directly from the code snippet, the result generated by LLMs is acceptable.

2. The few-shot covers only the first semantic part of the actual summary. The latter half, "from a set of coefficients" (highlighted in red in Fig. 5(b)), does not appear in the code summary. Nevertheless, given that "coefficients" is a key parameter of the function, its inclusion in the summary is reasonable.

```
def swap_slot_slot(self, resource_group_name, name, slot, target_slot, preserve_vnet,
                   custom_headers=None, raw=False, polling=True, **operation_config):
    ......
    lro_delay = operation_config.get(
        'long_running_operation_timeout',
        self.config.long_running_operation_timeout)
    if polling is True:
        polling_method = ARMPolling(lro_delay, **operation_config)
    elif polling is False:
        polling_method = NoPolling()
    else:
        polling_method = polling
    return LROPoller(self._client, raw_result, get_long_running_output, polling_method)
```

(a) A code snippet from the test set

```
#Ground-truth: Swaps two deployment slots of an app .
#Zero-shot: The code snippet for each slot , it creates a new slot and swaps it with the production slot.
#Few-shot: Swap the slot of a resource.
#EP4CS: Swaps two deployment slots of an app .
```

(b) Summaries generated by different method

Fig. 6. Code summarization case from CSN-Python No. 1820

3. Observing the results from EP4CS, it not only encompasses the entire semantic range of the actual summary but also captures details like "coefficients". EP4CS shows superior performance in adapting LLMs to the task of code summarization, thanks to the collaborative work of *Mapper* and *Struct-Agent*. However, upon closer examination of the generated content, we note some inappropriate expressions, such as the use of "associates". This indicates areas where EP4CS can still improve. Moving forward, we will continue to explore more effective ways to connect codePLMs with LLMs to better leverage the knowledge from codePLMs in assisting LLMs in producing high-quality code summaries.

Fig. 6 illustrates another representative case. Due to the code's length, only a portion is displayed here, while the full version is available in the CNS-python dataset (No. 1820). The reference summary for this code is: "Swaps two deployment slots of an app". Keywords such as "swap" and "slot" are explicitly present in the code, whereas terms like "deployment" and "app" are not directly mentioned. Nevertheless, this implicit information is crucial for understanding the code's real-world functionality. Consequently, LLMs encounter significant challenges when generating accurate code summarization.

Compared to the ground-truth summary, the analysis is as follows:

1. The zero-shot generated summary includes keywords like "swaps it" and "slot", but inaccurately states "creates a new slot". In reality, the code swaps two existing deployment slots without creating a new one. Thus, the use of "creates" is misleading.

2. The few-shot generated summary accurately conveys "swap the slot", but the phrase "of a resource" is overly vague and fails to specify that the operation involves swapping two deployment slots. Furthermore, the overall summary is too brief, potentially leading users to believe that only a single slot is involved, thereby misrepresenting the code's functionality.

3. Empirical observation of EP4CS demonstrates that its generated summary exhibits an exact correspondence with the reference summary. It correctly captures critical implicit elements such

as "deployment" and "app", which are not explicitly present in the code. Our analysis indicates that this implicit knowledge is derived from codePLM, which acquires and internalizes domain knowledge from extensive code corpora during training. Through the collaboration of the *Mapper* and *Struct-Agent*, this knowledge is extracted and integrated into soft prompts, which are then used to guide the LLM in generating summaries.

6.2 Threats to Validity

Internal Validity: Our experimental results are constrained by the evaluation metrics used. While we employed widely-used automated metrics such as BLEU and ROUGE-L, these primarily measure surface-level similarity between the generated text and reference texts[41, 43, 54]. For example, a generated summary may be grammatically correct but miss critical code logic, a flaw that current metrics struggle to detect. We mitigated this phenomenon by manual evaluation. Variations in instruction selection can impact model performance. To reduce this risk, we evaluated the effectiveness of different instructions on a small subset of the training data and followed recommendations from previous studies to identify the most suitable instructions.
External Validity: The collection of enhanced knowledge sets is constrained by the limitations of open-source resources, making it difficult to continue scaling. However, our framework effectively simulates this background knowledge, mitigating this external challenge. Furthermore, many studies have shown that the quality of real-world code summaries is often low, and they can even lead to misunderstandings, which in turn affect both model training and final evaluations. We have confirmed this issue through our experiments and discussions. Despite these findings, current research still lacks viable solutions, indicating the need for further improvements and time to address this problem.

7 Conclusion

We have proposed a novel framework for Code Summarization named EP4CS. EP4CS features a *Mapper* for background knowledge transformation and a *Struct-Agent*, which can induce an LLM to complete code summarization tasks by generating prompts enriched with prior knowledge. This framework integrates the advantages of both discrete and continuous prompting frameworks, achieving optimal performance without the need for complex directive design, and it also has a lower training cost.

Experimental results show that EP4CS is an effective code summarization technique and significantly outperforms existing technologies. As mentioned earlier, our proposed framework is highly scalable and does not rely on specific LLMs. Based on the results of RQ2, we speculate that substituting with more advanced large language models, could further improve code summarization performance. Meanwhile, due to resource limitations, although we report several advanced LLMs in terms of performance, the scope involved still lacks representativeness, and there is a shortage of research on larger-scale parameter models. This will also be part of our future research plans.

8 Data Availability

Our experimental materials are available at https://github.com/yuanxing2/EP4CS.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work is sponsored by the National Natural Science Foundation of China (Nos. 61973180, 62172249, 62202253, 62272254), the Natural Science Foundation of Shandong Province (Nos. ZR2024QF199, ZR2021MF092, ZR2021QF074, ZR2019MF014, ZR2019MF033, ZR2022MF326).

References

[1] Toufique Ahmed, Premkumar Devanbu. 2023. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 177, 5 pages. doi:10.1145/3551349.3559555

[2] Toufique Ahmed, Kunal Suresh Pai, Premkumar T. Devanbu, Earl T. Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 220:1–220:13. doi:10.1145/3597503.3639183

[3] Miltiadis Allamanis, Hao Peng, Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[4] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang et al. 2023. Qwen Technical Report. arXiv:2309.16609 [cs.CL]

[5] Satanjeev Banerjee, Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*, Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare R. Voss (Eds.). Association for Computational Linguistics, 65–72. https://aclanthology.org/W05-0909/

[6] Paheli Bhattacharya, Manojit Chakraborty, Kartheek N. S. N. Palepu, Vikas Pandey, Ishan Dindorkar, Rakesh Rajpurohit, Rishabh Gupta. 2023. Exploring Large Language Models for Code Explanation. In *Working Notes of FIRE 2023 - Forum for Information Retrieval Evaluation (FIRE-WN 2023), Goa, India, December 15-18, 2023 (CEUR Workshop Proceedings, Vol. 3681)*, Kripabandhu Ghosh, Thomas Mandl, Prasenjit Majumder, and Mandar Mitra (Eds.). CEUR-WS.org, 718–723. https://ceur-ws.org/Vol-3681/T7-15.pdf

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374

[8] Junyan Cheng, Iordanis Fostiropoulos, Barry W. Boehm. 2021. GN-Transformer: Fusing Sequence and Graph Representation for Improved Code Summarization. *CoRR* abs/2111.08874 (2021). arXiv:2111.08874

[9] Wei Cheng, Yuhan Wu, Wei Hu. 2024. Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 7957–7977. doi:10.18653/V1/2024.ACL-LONG.431

[10] Sergio Cozzetti B de Souza, Nicolas Anquetil, Káthia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. 68–75.

[11] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu et al. 2024. A Survey on In-context Learning. arXiv:2301.00234 [cs.CL] https://arxiv.org/abs/2301.00234

[12] Brian P. Eddy, Jeffrey A. Robinson, Nicholas A. Kraft, Jeffrey C. Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*. 13–22. doi:10.1109/ICPC.2013.6613829

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. doi:10.18653/V1/2020.FINDINGS-EMNLP.139

[14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=hQwb-lbM6EL

[15] Xue-Yong Fu, Md. Tahmid Rahman Laskar, Elena Khasanova, Cheng Chen, Shashi Bhushan TN. 2024. Tiny Titans: Can Smaller Large Language Models Punch Above Their Weight in the Real World for Meeting Summarization?. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Track, NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, Yi Yang, Aida Davani, Avi Sil, and Anoop Kumar (Eds.). Association for Computational Linguistics, 387–394. doi:10.18653/V1/2024.NAACL-INDUSTRY.33

[16] Yuexiu Gao, Chen Lyu. 2022. M2TS: multi-scale multi-modal approach based on transformer for source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, Ayushi Rastogi, Rosalia Tufano, Gabriele Bavota, Venera Arnaoudova, and Sonia Haiduc (Eds.).

ACM, 24–35. doi:10.1145/3524610.3527907

[17] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[18] Sina Gholamian, Paul A. S. Ward. 2021. On the Naturalness and Localness of Software Logs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 155–166. doi:10.1109/MSR52588.2021.00028

[19] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, Zenglin Xu. 2022. Source Code Summarization with Structural Relative Position Guided Transformer. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 13–24. doi:10.1109/SANER53432.2022.00013

[20] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. doi:10.18653/V1/2022.ACL-LONG.499

[21] Sonia Haiduc, Jairo Aponte, Laura Moreno, Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *2010 17th Working Conference on Reverse Engineering*. 35–44. doi:10.1109/WCRE.2010.13

[22] Rajarshi Haldar, Julia Hockenmaier. 2024. Analyzing the Performance of Large Language Models on Code Summarization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, Nicoletta Calzolari, Min-Yen Kan, Véronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue (Eds.). ELRA and ICCL, 995–1008. https://aclanthology.org/2024.lrec-main.89

[23] Wenpin Hou, Zhicheng Ji. 2024. A systematic evaluation of large language models for generating programming code. *CoRR* abs/2403.00894 (2024). doi:10.48550/ARXIV.2403.00894 arXiv:2403.00894

[24] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, Thomas Zimmermann. 2022. Practitioners' expectations on automated code comment generation. In *Proceedings of the 44th International Conference on Software Engineering*. 1693–1705.

[25] Dawei Huang, Chuan Yan, Qing Li, Xiaojiang Peng. 2024. From Large Language Models to Large Multimodal Models: A Literature Review. *Applied Sciences* 14, 12 (2024), 5068.

[26] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[27] Md. Ashraful Islam, Mohammed Eunus Ali, Md. Rizwan Parvez. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 4912–4944. doi:10.18653/V1/2024.ACL-LONG.269

[28] Xuehai Jia, Junwei Du, Minying Fang, Hao Liu, Yuying Li, Feng Jiang. 2025. Vulnerability detection with graph enhancement and global dependency representation learning. *Autom. Softw. Eng.* 32, 1 (2025), 14. doi:10.1007/S10515-024-00484-3

[29] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[30] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1646–1656. doi:10.1145/3611643.3613892

[31] Xin Jin, Jonathan Larson, Weiwei Yang, Zhiqiang Lin. 2023. Binary Code Summarization: Benchmarking ChatGPT/GPT-4 and Other Large Language Models. *CoRR* abs/2312.09601 (2023). doi:10.48550/ARXIV.2312.09601 arXiv:2312.09601

[32] Rafael-Michael Karampatsis, Charles Sutton. 2019. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. *CoRR* abs/1903.05734 (2019). arXiv:1903.05734 http://arxiv.org/abs/1903.05734

[33] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, Ashish Sabharwal. 2023. Decomposed Prompting: A Modular Approach for Solving Complex Tasks. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=_nGgzQjzaRy

[34] Jiachun Li, Pengfei Cao, Yubo Chen, Kang Liu, Jun Zhao. 2024. Towards Faithful Chain-of-Thought: Large Language Models are Bridging Reasoners. *CoRR* abs/2405.18915 (2024). doi:10.48550/ARXIV.2405.18915 arXiv:2405.18915

[35] Junnan Li, Dongxu Li, Silvio Savarese, Steven C. H. Hoi. 2023. BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models. In *International Conference on Machine Learning, ICML*

2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202), Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 19730–19742. https://proceedings.mlr.press/v202/li23q.html

[36] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim et al. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=KoFOg41haE

[37] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL] https://arxiv.org/abs/2305.06161

[38] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, Jie Tang. 2021. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602* (2021).

[39] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, Jie Tang. 2023. GPT understands, too. *AI Open* (2023).

[40] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.).

[41] Antonio Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, Gabriele Bavota. 2023. Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (2023), 2694–2706. https://api.semanticscholar.org/CorpusID:266551536

[42] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, Jianfeng Gao. 2024. Large language models: A survey. *arXiv preprint arXiv:2402.06196* (2024).

[43] Debanjan Mondal, Abhilasha Lodha, Ankita Sahoo, Beena Kumari. 2023. Robust Code Summarization. In *Proceedings of the 1st GenBench Workshop on (Benchmarking) Generalisation in NLP* (Singapore, 2023-12), Dieuwke Hupkes, Verna Dankers, Khuyagbaatar Batsuren, Koustuv Sinha, Amirhossein Kazemnejad, Christos Christodoulopoulos, Ryan Cotterell, and Elia Bruni (Eds.). Association for Computational Linguistics, 65–75. doi:10.18653/v1/2023.genbench-1.5

[44] OpenAI. 2022. *Codex Model*. https://beta.openai.com/docs/models/codexseriesprivate-beta

[45] Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. doi:10.3115/1073083.1073135

[46] Ajay Patel, Bryan Li, Mohammad Sadegh Rasooli, Noah Constant, Colin Raffel, Chris Callison-Burch. 2023. Bidirectional Language Models Are Also Few-shot Learners. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=wCFB37bzud4

[47] Chanathip Pornprasit, Chakkrit Tantithamthavorn. 2024. GPT-3.5 for Code Review Automation: How Do Few-Shot Learning, Prompt Design, and Model Fine-Tuning Impact Their Performance? *arXiv preprint arXiv:2402.00905* (2024).

[48] Bibek Poudel, Adam Cook, Sekou Traore, Shelah Ameli. 2024. DocuMint: Docstring Generation for Python using Small Language Models. *arXiv preprint arXiv:2405.10243* (2024).

[49] Alec Radford, Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training. https://api.semanticscholar.org/CorpusID:49313245

[50] Nils Reimers, Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3980–3990. doi:10.18653/V1/D19-1410

[51] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th international conference on software engineering*. 1597–1608.

[52] Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Yudu You, Tingxu Han, Yifei Ge, Yuling Hu, Bin Luo et al. 2024. An Extractive-and-Abstractive Framework for Source Code Summarization. *ACM Trans. Softw. Eng. Methodol.* 33, 3 (2024), 75:1–75:39. doi:10.1145/3632742

[53] Weisong Sun, Chunrong Fang, Yudu You, Yuchen Chen, Yi Liu, Chong Wang, Jian Zhang, Quanjun Zhang, Hanwei Qian, Wei Zhao et al. 2023. A Prompt Learning Framework for Source Code Summarization. *arXiv preprint arXiv:2312.16066* (2023). doi:10.48550/ARXIV.2305.12865

[54] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang et al. 2023. Automatic code summarization via chatgpt: How far are we? *arXiv preprint arXiv:2305.12865* (2023). doi:10.48550/ARXIV.2305.12865

[55] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, Zhenyu Chen. 2024. Source Code Summarization in the Era of Large Language Models. *CoRR* abs/2407.07959 (2024). doi:10.48550/ARXIV.2407.07959 arXiv:2407.07959

[56] Ilya Sutskever, Oriol Vinyals, Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (Eds.). 3104–3112.

[57] Tree-sitter. 2024. Tree-sitter: A parser generator tool and an incremental parsing library. https://github.com/tree-sitter/tree-sitter Accessed: 2024-07-15.

[58] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 382–394.

[59] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 5–16. doi:10.1109/ICSE48619.2023.00013

[60] Yue Wang, Weishi Wang, Shafiq R. Joty, Steven C. H. 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, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. doi:10.18653/V1/2021.EMNLP-MAIN.685

[61] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.).

[62] Edmund Wong, Taiyue Liu, Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 380–389. doi:10.1109/SANER.2015.7081848

[63] Hongqiu Wu, Hai Zhao, Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 1078–1090. doi:10.18653/V1/2021.FINDINGS-ACL.93

[64] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Trans. Software Eng.* 44, 10 (2018), 951–976. doi:10.1109/TSE.2017.2734091

[65] Hang Xu, Ning Kang, Gengwei Zhang, Chuanlong Xie, Xiaodan Liang, Zhenguo Li. 2021. NASOA: Towards Faster Task-oriented Online Fine-tuning with a Zoo of Models. In *2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021*. IEEE, 5077–5086. doi:10.1109/ICCV48922.2021.00505

[66] Kun Zhang, Zhendong Mao, Quan Wang, Yongdong Zhang. 2022. Negative-Aware Attention Framework for Image-Text Matching. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*. IEEE, 15640–15649. doi:10.1109/CVPR52688.2022.01521

[67] Mengmei Zhang, Mingwei Sun, Peng Wang, Shen Fan, Yanhu Mo, Xiaoxiao Xu, Hong Liu, Cheng Yang, Chuan Shi. 2024. GraphTranslator: Aligning Graph Model to Large Language Model for Open-ended Tasks. In *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, Tat-Seng Chua, Chong-Wah Ngo, Ravi Kumar, Hady W. Lauw, and Roy Ka-Wei Lee (Eds.). ACM, 1003–1014. doi:10.1145/3589334.3645682

[68] Fan Zhou, Chengtai Cao. 2021. Overcoming Catastrophic Forgetting in Graph Neural Networks with Experience Replay. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 4714–4722. doi:10.1609/AAAI.V35I5.16602

Received 2024-10-31; accepted 2025-03-31