Transforming Code Understanding: Clustering-Based Retrieval for Improved Summarization in Domain-Specific Languages

Baban Gain[1], **Dibyanayan Bandyopadhyay**[1], **Samrat Mukherjee**[1], **Aryan Sahoo**[1],

Saswati Dana[3], **Palanivel Kodeswaran**[3], **Sayandeep Sen**[3], **Asif Ekbal**[4], **Dinesh Garg**[3],

[1]Indian Institute of Technology Patna, [2]IBM Research, India

[2]Indian Institute of Technology Jodhpur

{gainbaban,dibyanayan,samratpisv123,aryansahoo.7277,asif.ekbal}@gmail.com

{sdana027,palani.kodeswaran,sayandes,garg.dinesh}@in.ibm.com

Abstract

A domain-specific extension of C language known as *extended Berkeley Packet Filter (eBPF)* has gained widespread acceptance for various tasks, including observability, security, and network acceleration in the cloud community. Due to its recency and complexity, there is an overwhelming need for natural language summaries of existing eBPF codes (particularly open-source code) for practitioners and developers, which will go a long way in easing the understanding and development of new code. However, being a niche Domain-Specific Language (DSL), there is a scarcity of available training data. In this paper, we investigate the effectiveness of LLMs for summarizing low-resource DSLs, in the context of eBPF codes. Specifically, we propose a clustering-based technique to retrieve in-context examples that are semantically closer to the test example and propose a very simple yet powerful prompt design that yields superior-quality code summary generation. Experimental results show that our proposed retrieval approach for prompt generation improves the eBPF code summarization accuracy up to 12.9 BLEU points over other prompting techniques. The codes are available at `https://github.com/babangain/ebpf_summ`.

1 Introduction

This paper addresses the highly industry-relevant challenge of *automatically generating summaries for code written in domain-specific languages (DSLs)*, where the availability of training data is often limited. While Large Language Models (LLMs) have shown remarkable progress in summarizing code written in widely-used programming languages (Liu et al., 2021; Zhang et al., 2020) like C, Java, and Python, their performance significantly drops when tasked with summarizing code in niche domain-specific languages—even when these DSLs are built on top of popular languages (as demonstrated in the experiments section).

Prominent examples of DSLs include CUDA (NVIDIA et al., 2020) and OpenGL (Woo et al., 1999), which extend C for specialized tasks such as GPU programming and graphics rendering. Although syntactically similar to their base language, these DSLs introduce unique control flow structures, domain-specific data manipulations, and low-level hardware interactions. Consequently, they pose challenges for both human developers and LLMs trying to analyze and understand this specialized code.

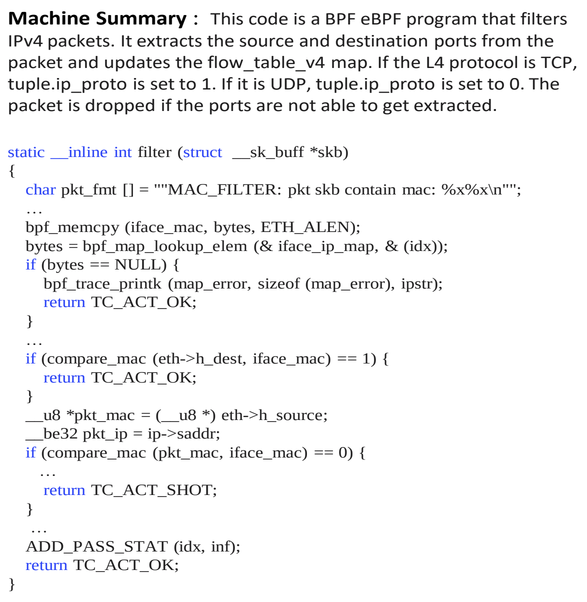
Customizing large language models (LLMs) for domain-specific languages (DSLs) is challenging due to limited training data. To address this, we use prompt engineering for domain adaptation rather than LLM-specific adjustments.

We found that enriching prompts with semantically similar (code, summary) examples improves LLM performance. To identify these examples, we introduce a clustering-based method that groups functionally similar codes using a distance metric. This approach enhances the accuracy of LLM-generated summaries during inference.

Our experiments demonstrate that even with a modest corpus of approximately 160 annotated eBPF code samples, our clustering-based approach significantly enhances the performance of the one-shot model, achieving a remarkable 300% improvement in BLEU score compared to the zero-shot baseline.

extended Berkeley Packet Filter (eBPF): eBPF is a rapidly growing technology (Vieira et al., 2020) that allows for the programmability of the Linux kernel, enabling the addition of new features and the optimization of existing kernel capabilities to meet specific business needs. Hyperscalers have widely adopted eBPF, with notable applications in observability and debugging (as seen with Netflix), networking (used by Meta, Cloudflare, and Google GKE), and security (integrated by Google K8s).

For a detailed tutorial on eBPF, we refer read-


Machine Summary : This code is a BPF eBPF program that filters IPv4 packets. It extracts the source and destination ports from the packet and updates the flow_table_v4 map. If the L4 protocol is TCP, tuple.ip_proto is set to 1. If it is UDP, tuple.ip_proto is set to 0. The packet is dropped if the ports are not able to get extracted.

```
static __inline int filter (struct  __sk_buff *skb)
{
   char pkt_fmt [] = ""MAC_FILTER: pkt skb contain mac: %x%x\n"";
   ...
   bpf_memcpy (iface_mac, bytes, ETH_ALEN);
   bytes = bpf_map_lookup_elem (& iface_ip_map, & (idx));
   if (bytes == NULL) {
      bpf_trace_printk (map_error, sizeof (map_error), ipstr);
      return TC_ACT_OK;
   }
   ...
   if (compare_mac (eth->h_dest, iface_mac) == 1) {
      return TC_ACT_OK;
   }
   __u8 *pkt_mac = (__u8 *) eth->h_source;
   __be32 pkt_ip = ip->saddr;
   if (compare_mac (pkt_mac, iface_mac) == 0) {
     ...
      return TC_ACT_SHOT;
   }
    ...
   ADD_PASS_STAT (idx, inf);
   return TC_ACT_OK;
}
```

Figure 1: A sample eBPF code with incorrect LLM (WizardCoder) generated summary

ers to the official documentation (eBPF). Figure 1 presents a code snippet of the eBPF function $filter()$ (eBPF), which checks whether a packet's MAC and IP addresses match those of the network interface and drops the packet if they do not. This code snippet highlights three key features of the eBPF language:

1) Hookpoint specificity: The Linux kernel exposes various hookpoints, such as tracepoints, function entry and exit, and packet reception, where eBPF code can be attached. Depending on the hookpoint, the input parameters and capabilities of the eBPF program vary. In this example, the code is attached to the TC hookpoint in the kernel's network stack.

2) eBPF helper functions: eBPF provides a set of specialized helper functions, such as $bpf\_redirect$, which are used within eBPF programs to interact with and modify kernel state.

3) eBPF maps: eBPF includes a mechanism called "maps" that facilitates data sharing between user-space and kernel-space, as well as between different kernel-space programs.

As seen in Figure 1, the eBPF code summary output by WizardCoder (Luo et al., 2023) for the $filter()$ function under zero-shot setting had no relevance to the given code snippet. We illustrate the limitations and challenges of current models in Table 3. These examples underscore the need for tailoring large language models (LLMs) to emerging domain-specific languages (DSLs), whose usage is rapidly expanding in niche fields, particularly in the domain of systems operations, where high performance, extensibility, and intelligent management are paramount.

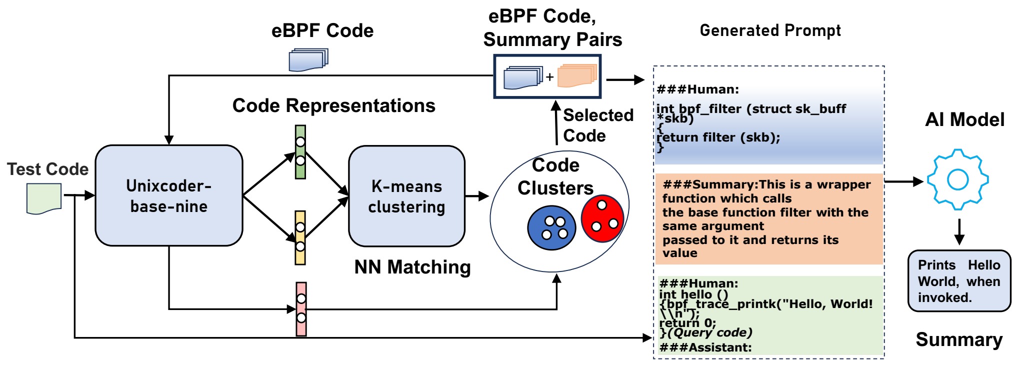
In this context, our prompting technique demonstrates *significantly superior performance*—a 300% improvement in code summarization compared to zero-shot baselines across multiple LLMs. Remarkably, this performance boost is achieved using a relatively small dataset consisting of 160 human-annotated functions. This highlights the potential of the proposed clustering-based approach to enhance LLM summarization capabilities for other DSLs, particularly in data-scarce environments.

Contributions: To the best of our knowledge, this work is the first to address the industry-critical problem of leveraging large language models (LLMs) for code summarization in low-resource languages, such as eBPF. While we focus on a single domain-specific language (DSL), we believe the approach is generalizable due to common characteristics shared by popular DSLs—namely, their derivation from widely-used programming languages with performant LLMs. Although experimenting with other DSLs is outside the scope of this work, our contributions are as follows:

i) We propose a clustering-based approach (Section 2.2) for selecting examples in in-context learning. Our results demonstrate that this approach improves performance across all tested models.
ii) We benchmark state-of-the-art code summarization models for eBPF code, evaluating both zero-shot and one-shot settings (Section 4). Notably, WizardCoder-15B achieves the best performance in the one-shot setting (see Table 1).
iii) We conduct a human evaluation of the generated summaries (Section 4.3) and publicly release the ratings, which can be leveraged for further tuning of generative models.

2 Methodology

eBPF codes are domain-specific, and thus, it is plausible that the language models are not trained on these eBPF codes. To benchmark their capabilities on eBPF summarization, we opt for two strategies: i) we consider zero-shot inference of decoder-only Large Language Models (LLMs) (Brown et al., 2020; Hoffmann et al., 2022). ii) we design a clustering strategy to prompt the LLMs with few-shot in-context examples, as discussed next.



```
###Human:
int bpf_filter (struct sk_buff
*skb)
{
return filter (skb);
}
```

```
###Summary:This is a wrapper
function which calls
the base function filter with the
same argument
passed to it and returns its
value
```

```
###Human:
int hello ()
{bpf_trace_printk("Hello, World!
\\n");
return 0;
}(Query code)
###Assistant:
```

Figure 2: **Prompting with "Nearest Code selection within the Cluster"**: For a query code at the test time, its cluster index is first determined, and the nearest code example for the test case is extracted from that cluster. Subsequently, the corresponding codes and annotations pairs are used as its one-shot in-context prompt. Prompt format given here is used for Codellama model.

2.1 Source-code Clustering

We begin by clustering a given set of eBPF source codes and utilizing these clusters to retrieve in-context few-shot examples for querying large language models (LLMs). In our scenario, the use of in-context prompts also reduces the reliance on a large quantity of domain-specific language (DSL) annotated data, which would otherwise be necessary for training the LLM. The steps of our clustering approach are detailed below.

Extracting Code Representations: Given a set of n eBPF code samples $(c_1, c_2, \ldots, c_n)$, we extract code-specific representations $(r_1, r_2, \ldots, r_n)$ by passing the samples through a feature extraction module, such as UnixCoder (Guo et al., 2022), which is trained on programming language tasks. These representations are obtained by applying average pooling to the final layer output of the model. This process is mathematically expressed as:

$$r_i = \mathrm{AvgPool}(\mathrm{UnixCoder}(c_i)) \quad (1)$$

Clustering: The code representations are then clustered using the k-means algorithm. To determine the optimal number of clusters, c, we employ the Elbow method (Kodinariya and Makwana, 2013), which helps identify the most appropriate value for k in k-means clustering.

Constructing In-Context Examples: Figure 2 illustrates the process of constructing in-context examples. At test time, we generate the code representation using the UnixCoder model. Applying k-means to this representation allows us to identify the corresponding cluster index, denoted by p. By iterating through the examples in cluster p and calculating their Euclidean distance from the test code's feature embedding (as defined in Equation 1). Euclidean distance is used in clustering because it effectively measures geometric proximity, enabling the grouping of similar data points while aligning naturally with variance-minimizing objectives like in k-means. We select the example with the smallest distance. This example is then used as a one-shot in-context prompt.

2.2 Prompt Design

To effectively leverage the capabilities of LLMs in inference mode, we adopt three distinct strategies for designing one-shot prompts.

Random One-Shot (S_{ro}): For each input code, we randomly select an example from the dataset and use the corresponding code-summary pair as the prompt for the model.

Random Code Selection within the Cluster (S_{rs}): For each test code, we extract its features using UnixCoder and determine its associated cluster. A random example from this cluster is then selected as the representative example to be used in the prompt.

Nearest Code Selection within the Cluster (S_{ncs}) This method, as detailed in Figure 2 and Section 2.1, involves finding the closest example within the cluster to which the test code is mapped. The nearest example is then used as the one-shot prompt.

In Section 4, we demonstrate the positive impact of our clustering methodology through both automatic and human evaluations, which show notable performance improvements across all models.

Model	Params	Zero-Shot					Random One-shot (S_{ro})				
		BLEU	Rouge-1	Rouge-2	Rouge-L	BERTScore	BLEU	Rouge-1	Rouge-2	Rouge-L	BERTScore
Deepseek-Coder	6.7B	3.6	0.2068	0.0406	0.1886	0.8304	1.2	0.1458	0.0179	0.1310	0.6856
Codellama-Instruct	7B	2.4	0.1992	0.0428	0.1820	0.7567	2.8	0.1336	0.0259	0.1197	0.5479
WizardCoder-Python	7B	4.0	**0.2260**	**0.0444**	**0.2020**	0.8405	2.9	**0.2165**	**0.0419**	**0.1953**	0.8380
Mistral-OpenOrca	7B	**4.1**	0.2240	0.0359	0.2008	0.8411	3.2	0.2055	0.0335	0.1814	**0.8491**
Zephyr-beta	7B	3.6	0.2150	0.0358	0.1899	**0.8416**	**3.9**	0.2116	0.0378	0.1849	0.8424
WizardCoder	15B	4.2	0.2352	0.0478	0.2095	0.8437	4.8	0.2216	0.0435	0.1961	0.8483

(a) Comparison of Zero-shot with Random one-shot based prompting.

Model	Params	Random Code selection within the Cluster (S_{rs})					Nearest Code selection within the Cluster (S_{ncs})				
		BLEU	Rouge-1	Rouge-2	Rouge-L	BERTScore	BLEU	Rouge-1	Rouge-2	Rouge-L	BERTScore
Deepseek-Coder	6.7B	2.3	0.1628	0.0333	0.1483	0.7331	7.2	0.2437	0.0774	0.2216	0.7650
Codellama-Instruct	7B	4.9	0.1540	0.0459	0.1393	0.5491	13.0	0.2146	0.0891	0.1952	0.8517
WizardCoder-Python	7B	3.6	**0.2351**	0.0517	**0.2117**	0.8387	5.9	0.2636	0.0697	0.2381	0.8477
Mistral-OpenOrca	7B	**5.1**	0.2242	0.0513	0.2006	0.8341	**14.7**	**0.3402**	**0.1378**	**0.3112**	**0.8701**
Zephyr-beta	7B	4.8	0.2316	**0.0541**	0.2086	**0.8404**	7.9	0.2804	0.0809	0.2474	0.8551
WizardCoder	15B	7.9	0.2803	0.0787	0.2520	0.8548	17.7	0.3509	0.1550	0.3210	0.8663

(b) Comparison of "Random Code selection within the Cluster" with "Nearest Code selection within the cluster"

Table 1: Comparison of different models based on automatic evaluation metrics. The top performing model within the 7B category is highlighted in **bold**. Overall, top performer is highlighted with underline

3 Experimental Setup

In the experimental setup, we utilize commented datasets for eBPF source code obtained from the eBPF-DevSecTools repository (eBPF, 2023). This comprehensive repository includes source code from various eBPF projects, such as notable ones like Cilium (Cilium, 2018) and Katran (Katran, 2018), as well as utility collections like BCC (bcc, 2015). For evaluation, we use SacreBLEU (Papineni et al., 2002; Post, 2018), reporting the geometric mean up to 4-grams.

Dataset: A total of 160 functions were annotated by students and professionals with sufficient domain knowledge and annotation proficiency. These functions were manually annotated with summaries at the function level, and in some cases, at the line level as well. We extracted features from the summaries using the Unixcoder-base-nine model and calculated pairwise similarity, ensuring a set of 136 deduplicated examples with an average summary length of 52 words per example.

4 Results

In Section 4.1, we present the overall performance through a quantitative evaluation of model outputs. Subsequently, in Section 4.2, we compare the performance of various prompting strategies. Finally, in Section 4.3, we present the results of a human evaluation conducted by experienced professionals on a subset of our dataset.

4.1 Quantitative Evaluation

Our experimental results, summarized in Table 1a, demonstrate the performance of recent large language models using zero-shot and one-shot prompts with various strategies. The WizardCoder-15B model (Luo et al., 2023) consistently outperforms the other models across all prompting strategies and evaluation metrics. Notably, performance improves from a BLEU score of 4.2 in the zero-shot setting to 4.8 with random one-shot prompting. Further enhancements are observed when employing S_{rs}, achieving a BLEU score of 7.9, with the highest BLEU score of 17.7 obtained using our proposed clustering technique, S_{ncs}.

In the zero-shot setting, the 7B models exhibit inconsistent performance across the various metrics. Zephyr-beta (Tunstall et al., 2023) ranks second in BERTScore, Mistral-OpenOrca (Mukherjee et al., 2023) achieves the second-highest BLEU score, while WizardCoder-Python 7B performs well on several ROUGE metrics. When utilizing S_{ro} or S_{rs} as prompting strategies, the quality of in-context examples leads to inconsistent performance across both models and metrics. The soft-prompt design employed by the S_{ncs} technique proves to be the most effective strategy, achieving a BLEU score of 17.7 and a BERTScore of 0.8663. Among the 7B models, Mistral-OpenOrca consistently performs well in the optimal one-shot setting, with CodeLlama-Instruct (Roziere et al.,

2023) ranking third in terms of BLEU score. In contrast, Deepseek-Coder (DeepSeek, 2023) does not demonstrate competitive performance compared to the other LLMs considered in this study.

4.2 Qualitative Insights into Prompting Strategies

In this section, we examine the effects of different prompting methods and model parameter adjustments, providing a comprehensive understanding of their impact on overall performance.

4.2.1 Zero-shot vs. S_{ncs}

Our qualitative case study reveals a clear distinction between zero-shot and contextual few-shot scenarios. In the absence of additional training examples (zero-shot conditions), the models, particularly *Codellama*, struggled with complex code structures, leading to a substantial number of instances where the models failed to generate any summaries. However, as we shifted to one-shot prompt, a significant improvement became evident. The inclusion of contextual code-summary pairs was crucial in addressing the issue of non-summary generation seen in zero-shot conditions.

Moreover, our study consistently observed performance improvements across all models, culminating in a remarkable 300% increase in BLEU score (from 4.2 in zero-shot to 17.7 in one-shot) when employing one-shot prompts. This enhancement highlights the positive effect of integrating contextual information and task-specific examples in improving the code summarization capabilities of language models. Examples of Zero-Shot vs. S_{ncs} are provided in Appendix E.

4.2.2 S_{ncs} vs. S_{ro}

While it is well-established that prompts can enhance the quality of generated outputs, the relevance and quality of the examples used in the prompt significantly affect the performance. When random examples from the dataset are employed, the quality of the generated summaries degrades substantially (e.g., Mistral-OpenOrca's BLEU score drops from 14.7 to 3.2), as shown in row 4 of Table 1a and Table 1b. Similar trends were observed across other models, underscoring the importance of selecting appropriate in-context examples for optimal results.

4.2.3 Zero-shot vs. S_{ro}

We observed mixed results when using random examples for one-shot prompts compared to zero-shot. For instance, Codellama-Instruct (row 1 of Table 1a) showed an improvement in BLEU score, while ROUGE and BERTScore declined. Although the BLEU score improved for the three models, it decreased for the other three. In the case of WizardCoder, the random one-shot strategy had a negative impact on the 7B models but yielded positive results for the 15B model, *suggesting that larger models exhibit better in-context learning capabilities*. Upon further inspection of randomly chosen examples, we found that models such as WizardCoder-7B, Deepseek-Coder, and Mistral often mimicked the patterns and phrases from the random one-shot examples, leading to a decrease in performance.

4.2.4 Effect of Model Parameters

We observe that larger models, such as WizardCoder-15B, demonstrate significant performance improvements with our proposed approach. For instance, the BLEU score increased from 4.2 in the zero-shot setting to 17.7 when using S_{ncs}. In contrast, the smaller WizardCoder-7B model saw only a modest improvement, with a BLEU score rising from 4.0 to 5.9. This suggests that larger models are more adept at capturing the properties of the code from the provided examples.

4.3 Manual Evaluation

Evaluating code summarization is inherently challenging due to the varying levels of granularity at which summaries can be written. In addition to automatic metric evaluations, we conducted a manual evaluation of the model-generated summaries produced by the proposed approach S_{ncs}, as detailed in Table 2. A total of 90 code-summary pairs were evaluated by four domain experts, resulting in 360 individual evaluations. To measure inter-rater agreement, 30 pairs were shared among the evaluators[1]. For each of the six LLMs, we randomly selected 15 data points from our test set along with their corresponding model-generated summaries, and these were provided to the experts for evaluation. The experts, each with over a year of experience in eBPF code, assessed both common and unique summaries. Specifically, 5 of the 15 summaries were shared across annotators, while the remaining 10 were distinct. Details on inter-rater agreement can be found in Appendix H, and the annotation guidelines are provided in Appendix A.

[1]The evaluators are experienced industry professionals with significant expertise in the task.

Model	Expert 1	Expert 2	Expert 3	Expert 4	Average
Codellama-Instruct	1.33	1.27	1.13	2.00	1.43
WizardCoder-Python [7B]	2.87	1.80	1.93	2.53	2.28
Deepseek-Coder	2.43	2.30	2.83	3.60	2.79
Mistral-OpenOrca	2.37	2.77	2.90	2.93	2.74
Zephyr-beta	3.20	2.77	3.00	3.10	**3.02**
WizardCoder [15B]	2.97	2.77	2.67	3.27	2.92

Table 2: Experts' ratings on 0 to 4 scale. Higher score indicates a better quality of summary

To eliminate potential bias, the annotators were presented with the generated summaries only, without any identifying information regarding the models, descriptions, or references. Summaries were rated on a scale from 0 to 4, with 0 representing the lowest score and 4 representing the highest. For each model, we calculated the average rating assigned by each expert, as shown in Table 2. The Zephyr-beta model emerged as the top performer, achieving an impressive average rating of 3.02 (out of 4). The WizardCoder-15B model followed closely with an average rating of 2.92, and the Deepseek-Coder-7B model ranked third with an average rating of 2.79.

Interestingly, when we compared these results with BLEU scores, the top three models in terms of BLEU were WizardCoder-15B, Mistral-OpenOrca-7B, and Codellama-Instruct. Despite its high BLEU score, Codellama-Instruct received the lowest average rating (1.43) from the manual evaluation, highlighting a significant discrepancy between automatic and human evaluations. This suggests that BLEU may not be a reliable metric for evaluating concise code summarization. Furthermore, BERTScore and ROUGE showed similar results for Codellama-Instruct-7B, despite its lower manual evaluation scores. These findings underscore the need for developing more reliable metrics that can better capture the nuances and quality of concise code summarization. Additional insights into challenging cases are provided in Appendix F.

5 Related Work

The evolution of code summarization, driven by (Haiduc et al., 2010)'s early work, initially focused on analyzing source code as text for generating objective-oriented programming language descriptions. Later, (Moreno et al., 2013) incorporated part-of-speech tagging but focused on keywords, overlooking control flows and data dependencies. Recently, LLMs (Feng et al., 2020; Guo et al., 2020; Ahmad et al., 2021; Wang et al., 2021; Cassano et al., 2024; Ahmed and Devanbu, 2023) have demonstrated significant progress in developing AI systems that solve a wide variety of code/programming language-related tasks as well. Sate-of-art LLMs (Radford et al., 2019; Wang and Komatsuzaki, 2021; Black et al., 2022) are able to perform well on natural language descriptions with minimal examples. Min et al. (2022) showed that using random labels instead of actual labels in in-context examples does not hurt performance by a large margin. Liu et al. (2022) use a based approach to select examples for in-context learning.

However, the existing labeled benchmark datasets for code summarization mainly originate from public repositories or coding competitions. To address this, we utilize human-annotated eBPF codes (eBPF, 2023) for explanation generation, marking a pioneering effort in generating explanations for eBPF code.

6 Conclusion

This paper brings attention to the critical issue of low performance exhibited by large language models (LLMs) in summarizing code written in low-resource domain-specific languages (DSLs), using eBPF as a candidate language. To the best of our knowledge, this is the first study to leverage LLMs for eBPF code summarization.

We propose a straightforward clustering-based technique to retrieve functionally similar code, which serves as in-context examples for effectively querying LLMs to generate eBPF code summaries. Experimental results demonstrate that our approach improves the summarization accuracy of various LLMs by 12.9 BLEU points over random one-shot examples.

While the results pertain specifically to eBPF, the shared characteristics of popular DSLs, such as their derivation from mainstream programming languages (for which performant LLMs exist), provide confidence in the generalizability of the approach. Experimentation with other DSLs is beyond the scope of the current work and is part of our planned future research.

Acknowledgment

We sincerely thank Dushyant Behl and Sachee Mishra for their invaluable contributions in manually rating the quality of code summaries, which significantly enhanced the evaluation process for this work. Baban Gain and Dibyanayan Bandyopadhyay gratefully acknowledge the Prime Minister's Research Fellowship (PMRF) program for providing financial support and enabling this research.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Toufique Ahmed and 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*, ASE '22, New York, NY, USA. Association for Computing Machinery.

bcc. 2015. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

Sidney Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, Usvsn Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pages 95–136, virtual+Dublin. Association for Computational Linguistics.

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

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proc. ACM Program. Lang.*, 8(OOPSLA2).

Cilium. 2018. Cilium : ebpf-based networking, security, and observability. https://github.com/cilium/cilium/.

DeepSeek. 2023. Deepseek coder: Let the code write itself. https://github.com/deepseek-ai/DeepSeek-Coder.

eBPF. ebpf documentation. https://ebpf.io/what-is-ebpf/.

eBPF. ebpf filter.

eBPF. 2023. ebpf-projects-annotations. https://github.com/eBPFDevSecTools/annotations.

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. *arXiv preprint arXiv:2002.08155*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. *17th Working Conference on Reverse Engineering*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Katran. 2018. katran : A high performance layer 4 load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

Trupti M. Kodinariya and Prashant R. Makwana. 2013. Review on determining number of cluster in k-means clustering.

Terry K Koo and Mae Y Li. 2016. A guideline of selecting and reporting intraclass correlation coefficients for reliability research. *Journal of chiropractic medicine*, 15(2):155–163.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online. Association for Computational Linguistics.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2021. Retrieval-augmented generation for code summarization via hybrid gnn. *Preprint*, arXiv:2006.05405.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. 2023. Orca: Progressive learning from complex explanation traces of gpt-4. *arXiv preprint arXiv:2306.02707*.

NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. Cuda, release: 10.2.89.

Kishore Papineni, Salim Roukos, Todd Ward, and 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*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Matt Post. 2018. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Brussels, Belgium. Association for Computational Linguistics.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

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

C. Spearman. 1904. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101.

Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Kashif Rasul, Younes Belkada, Shengyi Huang, Leandro von Werra, Clémentine Fourrier, Nathan Habib, et al. 2023. Zephyr: Direct distillation of lm alignment. *arXiv preprint arXiv:2310.16944*.

Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1).

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. `https://github.com/kingoflolz/mesh-transformer-jax`.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. 1999. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1385–1397, New York, NY, USA. Association for Computing Machinery.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

A Annotation Guidelines

The following guidelines were provided to the experts during the human evaluation of the outputs.

- 4=> The summary describes the code perfectly, and there is no unrelated information in the summary. If one looks at the summary, s/he will understand what the code is doing without looking at the code.
- 3=> The summary describes the code very well, missing only some trivial points or containing some extra trivial information.

Function Name	Description	Model Query	Mistral-Openorca	Zephyr	Wizardcoder
XDP_DROP	XDP_DROP is a helper for pkt_stop_processing_drop_packet. When XDP_DROP is returned, it indicates that the XDP program has chosen to drop the packet, preventing it from continuing through the regular network stack for additional processing.	Given a code containing XDP_DROP, is XDP_DROP related to pkt_stop_processing_drop_packet in any way?	Yes	No	No
bpf_map_lookup_elem	In eBPF, bpf_map_lookup_elem is a helper function used to retrieve values associated with keys in BPF maps. It facilitates efficient access and manipulation of shared data between eBPF programs and the kernel or user space.	Is bpf_map_lookup_elem helper function related to map_read in any way?	No	Yes	No
bpf_redirect	In eBPF, bpf_redirect is a helper function that redirects packets to a specified network interface. It enables custom packet forwarding or load balancing within eBPF programs, allowing them to influence packet routing in the network stack.	Is bpf_redirect helper function related to pkt_alter_or_redo_processing_or_interface in any way?	Yes	No	Yes

Table 3: Examples showing the limitations of Large Language Models in understanding domain-specific details; We prompt the models with some domain-specific functions and their related functions/variables and ask ("Model Query" column) if they are related. Note that the actual answer to all the questions is "Yes". However, the models are generating "No" indicating that these models do not have any knowledge of the internal workings of DSLs.

- 2=> The summary is good, but it is difficult to understand after reading it once or twice. The summary is wrong at one or two points but not too critical. The readers need to read it multiple times or look at the code thoroughly to understand it.
- 1=> The summary is on a similar topic to the code, but it misunderstood what the code is doing (i.e., the logic is explained wrongly in the summary)
- 0=> The summary is not at all related to the code/ Summary not generated at all
- Use ratings of 3.5, 2.5, 1.5, and 0.5 for summaries that do not belong to the aforementioned categories.

A.1 Challenges of using LLMs for domain-specific query

Unfortunately, traditional code summarization models are not well-suited for summarizing eBPF codes due to the complexity of eBPF codes, limited understanding of kernel concepts, and data sparsity. Particularly, the users of domain-specific extension languages have different expectations from the LLMs compared to their base PLs.

B Results with Two-Shot prompts

We investigate the effectiveness of a clustering method using two-shot prompts. Our experiments involve two models: WizardCoder-15B, which achieves the highest automatic metric scores in the one-shot setting, and Zephyr-beta-7B, which receives the best manual ratings. [2] From Table 4, we observe consistent improvements in (S_{ncs}) compared to (S_{rs}). However, the gains with two shots are minimal compared to one (Table 1) shot in (S_{rs}). We observed a slight decline in the BLEU and Rouge-2 compared to (S_{ncs}), which is likely due to the fact that the additional example used as an in-context example is not as relevant as the nearest example.

C Models

In recent times, LLMs like ChatGPT have garnered significant attention. Various LLMs have been developed and trained on programming languages and natural language datasets. These models are readily applicable for inference without additional modifications. We employ them in inference mode with a one-shot example utilizing both prompts with random examples and dynamic in-context prompts generated obtained via clustering.

- **Codellama:** Codellama is a fine-tuned version of Llama2 having infilling capabilities, zero-shot instruction following ability, as well as support for large input contexts for programming tasks.
- **WizardCoder:** Similar to Codellama, WizardCoder is obtained from Llama2, and it has similar capabilities. Unlike other major code LLMs, WizardCoder is trained with code-specific instructions.
- **Deepseek-Coder:** The Deepseek-Coder model

[2] In cases where the maximum length limit was exceeded for two-shot prompts (observed in two examples), we used the corresponding outputs from the one-shot setting under identical experimental conditions.

Model	Params	Random Code selection within the Cluster (S_{rs})					Nearest Code selection within the Cluster (S_{ncs})				
		BLEU	Rouge-1	Rouge-2	Rouge-L	BERTScore	BLEU	Rouge-1	Rouge-2	Rouge-L	BERTScore
Zephyr-beta	7B	4.7	0.2287	0.0498	0.2071	0.8450	6.9	0.2615	0.0695	0.2279	0.8510
WizardCoder	15B	9.2	0.2828	0.0823	0.2565	0.8551	**17.1**	**0.3562**	**0.1514**	**0.3247**	**0.8673**

Table 4: Comparison of "Random Code selection within the Cluster" with "Nearest Code selection within the cluster" on Two-shot prompts

is pre-trained on 2 Trillion tokens over more than 80 programming languages. The training data consists of 87% code and 13% natural language text. Further, it was fine-tuned on 2B tokens of instructions.

- **Mistral-OpenOrca:** Mistral-OpenOrca is obtained by fine-tuning Mistral-7B with Openorca, which is a dataset containing instructions.
- **Zephyr-beta:** Zephyr-beta is also a fine-tuned version of Mistral. It was trained on publicly available as well as synthetic datasets using Direct Preference Optimization (DPO).

D Prompt Template

```
Below is an instruction that describes a task.
     Write a response that appropriately
     completes the request.

### Instruction:
Generate a short and concise summary for the
     following code. Do not refer to the example
     code in the generated summary. The first
     code is only for example. Code: {Example
     code}
Summary: {Summary of the Example code}
Now, summarize the following. Code: {Current
     code}
Summary:

### Response:
```

Example 1: This represents the prompt format we used for WizardCoder models. We use similar prompt formats for other models with corresponding instruction templates.

E Code Examples

In this section, we present three distinct eBPF codes, featured in Table 5 for the comparison of generated summary between 0-shot and our proposed approach, S_{ncs}. The selection of these codes aims to showcase our experimentation on both larger and smaller codebases, illustrating that our approach consistently yields superior results across all cases.

- **Code ID: D1** - *ARP Handling Code*
  - Project Name: cilium [3]

```
int tail_handle_arp (struct __ctx_buff
     *ctx) { union macaddr mac = NODE_MAC;
     union macaddr smac; struct trace_ctx
     trace = { .reason =
     TRACE_REASON_CT_REPLY, .monitor =
     TRACE_PAYLOAD_LEN,} ; __be32 sip;
     __be32 tip; int ret; struct
     bpf_tunnel_key key = {} ; struct
     vtep_key vkey = {} ; struct
     vtep_value *info; if (unlikely
     (ctx_get_tunnel_key (ctx, &key,
     sizeof (key), 0) < 0)) return
     send_drop_notify_error (ctx, 0,
     DROP_NO_TUNNEL_KEY, CTX_ACT_DROP,
     METRIC_INGRESS); if (!arp_validate
     (ctx, &mac, &smac, &sip, &tip) ||
     !__lookup_ip4_endpoint (tip)) goto
     pass_to_stack; vkey.vtep_ip = sip &
     VTEP_MASK; info = map_lookup_elem (&
     VTEP_MAP, & vkey); if (!info) goto
     pass_to_stack; ret =
     arp_prepare_response (ctx, & mac,
     tip, & smac, sip); if (unlikely (ret
     != 0)) return send_drop_notify_error
     (ctx, 0, ret, CTX_ACT_DROP,
     METRIC_EGRESS); if
     (info->tunnel_endpoint) return
     __encap_and_redirect_with_nodeid
     (ctx, info->tunnel_endpoint,
     SECLABEL, WORLD_ID, &trace); return
     send_drop_notify_error (ctx, 0,
     DROP_UNKNOWN_L3, CTX_ACT_DROP,
     METRIC_EGRESS); pass_to_stack :
     send_trace_notify (ctx,
     TRACE_TO_STACK, 0, 0, 0,
     ctx->ingress_ifindex, trace.reason,
     trace.monitor); return CTX_ACT_OK; }
```

- **Code ID: D2** - *IPv4 Hashing Code*
  - Project Name: suricata [4]

```
static __always_inline int ipv4_hash
     (struct __sk_buff *skb) { __u32
     nhoff; __u32 src, dst; nhoff =
     skb->cb[0]; src = load_word (skb,
     nhoff + offsetof (struct iphdr,
     saddr)); dst = load_word (skb, nhoff
     + offsetof (struct iphdr, daddr));
     #if 0 char fmt [] = "Got addr: %x ->
     %x at %d\n"; bpf_trace_printk (fmt,
     sizeof (fmt), src, dst, nhoff);
```

[3] https://github.com/cilium/cilium/blob/8f606bafb696076609a01f88204d895576043950/bpf/bpf_overlay.c#L353

[4] https://github.com/OISF/suricata/blob/master/ebpf/lb.c#L42

Code ID	D.1	D.2	D.3
Model	Mistral	CodeLlama	Mistral
zero-shot	-	The code is a BPF program that hashes the source and destination IP addresses of an IPv4 packet.	#endif /* ENABLE_ROUTING */ }
S_{ncs}	Handle ARP packets and prepare response for ARP packets.	This code defines a function called `ipv4_hash` that takes a `struct __sk_buff *skb` as an argument. The function uses the source and destination IP addresses of the packet, which are loaded from the `saddr` and `daddr` fields of the IP header, respectively. The function then returns the sum of the source and destination IP addresses as the final hash value. The code also includes a comment that prints a trace message to the kernel log using `bpf_trace_printk`.	Handling egress ipv6 traffic. It marks the traffic if it has a remote endpoint destination and restores ct_state from per packet lb handling in the previous tail call. Also enabling the per packet load balancing in the previous tail call.

Table 5: Comparison between summary generated in Zero-shot and "Nearest Code selection within the Cluster" for the respective code ID given in the columns.

```
#endif return src + dst; }
```

- **Code ID: D3** - *egress IPv6 Code*
  - Project Name: cilium [5]

```
static __always_inline int
    handle_ipv6_from_lxc (struct
    __ctx_buff *ctx, __u32 *dst_id) {
    struct ct_state ct_state_on_stack
    __maybe_unused, *ct_state,
    ct_state_new = {}; struct
    ipv6_ct_tuple tuple_on_stack
    __maybe_unused, *tuple; #ifdef
    ENABLE_ROUTING union macaddr
    router_mac = NODE_MAC; #endif struct
    ct_buffer6 *ct_buffer; void *data,
    *data_end; struct ipv6hdr *ip6; int
    ret, verdict = 0, l4_off, hdrlen,
    zero = 0; struct trace_ctx trace = {
    .reason = TRACE_REASON_UNKNOWN,
    .monitor = 0,} ; __u32 __maybe_unused
    tunnel_endpoint = 0; __u8
    __maybe_unused encrypt_key = 0; enum
    ct_status ct_status; bool
    hairpin_flow = false; __u8
    policy_match_type =
    POLICY_MATCH_NONE; __u8 audited = 0;
    bool __maybe_unused dst_remote_ep =
    false; __u16 proxy_port = 0; bool
    from_l7lb = false; bool
    emit_policy_verdict = true; if
    (!revalidate_data (ctx, &data,
    &data_end, &ip6)) return DROP_INVALID;
if (1) { const union v6addr *daddr =
    (union v6addr *) &ip6->daddr; struct
    remote_endpoint_info *info; info =
    lookup_ip6_remote_endpoint (daddr);
    if (info && info->sec_label) {
    *dst_id = info->sec_label;
    tunnel_endpoint =
    info->tunnel_endpoint; encrypt_key =
    get_min_encrypt_key (info -> key);
    #ifdef ENABLE_WIREGUARD if
    (info->tunnel_endpoint != 0 &&
    !identity_is_node (info->sec_label))
    dst_remote_ep = true; #endif /*
    ENABLE_WIREGUARD */ } else { *dst_id
    = WORLD_ID; } cilium_dbg (ctx, info ?
    DBG_IP_ID_MAP_SUCCEED6 :
    DBG_IP_ID_MAP_FAILED6, daddr->p4,
    *dst_id); } #ifdef
    ENABLE_PER_PACKET_LB #if
    !defined(DEBUG) &&
    defined(TUNNEL_MODE) if
    (!revalidate_data (ctx, &data,
    &data_end, &ip6)) return
    DROP_INVALID; #endif
    lb6_ctx_restore_state (ctx,
    &ct_state_new, &proxy_port); #endif
    /* ENABLE_PER_PACKET_LB */ ct_buffer
    = map_lookup_elem (&
    CT_TAIL_CALL_BUFFER6, & zero); if
    (!ct_buffer) return
    DROP_INVALID_TC_BUFFER; if
    (ct_buffer->tuple.saddr.d1 == 0 &&
    ct_buffer->tuple.saddr.d2 == 0)
    return DROP_INVALID_TC_BUFFER; #if
    HAVE_DIRECT_ACCESS_TO_MAP_VALUES
    tuple = (struct ipv6_ct_tuple *)
    &ct_buffer->tuple; ct_state = (struct
    ct_state *) &ct_buffer->ct_state;
    #else memcpy (&tuple_on_stack,
    &ct_buffer->tuple, sizeof
    (tuple_on_stack)); tuple =
    &tuple_on_stack; memcpy
    (&ct_state_on_stack,
    &ct_buffer->ct_state, sizeof
    (ct_state_on_stack)); ct_state =
    &ct_state_on_stack; #endif /*
    HAVE_DIRECT_ACCESS_TO_MAP_VALUES */
    trace.monitor = ct_buffer->monitor;
    ret = ct_buffer->ret; ct_status =
    (enum ct_status) ret; trace.reason =
    (enum trace_reason) ret; #if
    defined(ENABLE_L7_LB) if (proxy_port
    > 0) { cilium_dbg3 (ctx, DBG_L7_LB,
    tuple->daddr.p4, tuple->saddr.p4,
    bpf_ntohs (proxy_port)); verdict =
    proxy_port; emit_policy_verdict =
    false; goto skip_policy_enforcement;
    } #endif /* ENABLE_L7_LB */ if
    ((ct_status == CT_REPLY || ct_status
    == CT_RELATED) &&
    ct_state->proxy_redirect) { return
    ctx_redirect_to_proxy6 (ctx, tuple,
    0, false); }
if (hairpin_flow) { emit_policy_verdict =
```

[5] https://github.com/cilium/cilium/blob/main/bpf/bpf_lxc.c#L389

```
    false; goto skip_policy_enforcement;
    } verdict = policy_can_egress6 (ctx,
    tuple, SECLABEL, * dst_id, &
    policy_match_type, & audited); if
    (ct_status != CT_REPLY && ct_status
    != CT_RELATED && verdict < 0) {
    send_policy_verdict_notify (ctx,
    *dst_id, tuple->dport,
    tuple->nexthdr, POLICY_EGRESS, 1,
    verdict, policy_match_type, audited);
    return verdict; }
    skip_policy_enforcement : #if
    defined(ENABLE_L7_LB) from_l7lb =
    ctx_load_meta (ctx, CB_FROM_HOST) ==
    FROM_HOST_L7_LB; #endif switch
    (ct_status) { case CT_NEW : if
    (emit_policy_verdict)
    send_policy_verdict_notify (ctx,
    *dst_id, tuple->dport,
    tuple->nexthdr, POLICY_EGRESS, 1,
    verdict, policy_match_type, audited);
    ct_recreate6 :
    ct_state_new.src_sec_id = SECLABEL;
    ret = ct_create6 (get_ct_map6
    (tuple), & CT_MAP_ANY6, tuple, ctx,
    CT_EGRESS, & ct_state_new, verdict >
    0, from_l7lb); if (IS_ERR (ret))
    return ret; trace.monitor =
    TRACE_PAYLOAD_LEN; break; case
    CT_REOPENED : if
    (emit_policy_verdict)
    send_policy_verdict_notify (ctx,
    *dst_id, tuple->dport,
    tuple->nexthdr, POLICY_EGRESS, 1,
    verdict, policy_match_type, audited);
    case CT_ESTABLISHED : if (unlikely
    (ct_state->rev_nat_index !=
    ct_state_new.rev_nat_index)) goto
    ct_recreate6; break; case CT_RELATED
    : case CT_REPLY : policy_mark_skip
    (ctx); hdrlen = ipv6_hdrlen (ctx, &
    tuple -> nexthdr); if (hdrlen < 0)
    return hdrlen; l4_off = ETH_HLEN +
    hdrlen; #ifdef ENABLE_NODEPORT #
    ifdef ENABLE_DSR if (ct_state->dsr) {
    ret = xlate_dsr_v6 (ctx, tuple,
    l4_off); if (ret != 0) return ret; }
else # endif /* ENABLE_DSR */ if
    (ct_state->node_port) {
    send_trace_notify (ctx,
    TRACE_TO_NETWORK, SECLABEL, *dst_id,
    0, 0, trace.reason, trace.monitor);
    ctx->tc_index |=
    TC_INDEX_F_SKIP_RECIRCULATION;
    ep_tail_call (ctx,
    CILIUM_CALL_IPV6_NODEPORT_REVNAT);
    return DROP_MISSED_TAIL_CALL; }
    #endif /* ENABLE_NODEPORT */ if
    (ct_state->rev_nat_index) { struct
    csum_offset csum_off = {} ;
    csum_l4_offset_and_flags
    (tuple->nexthdr, &csum_off); ret =
    lb6_rev_nat (ctx, l4_off, & csum_off,
    ct_state -> rev_nat_index, tuple, 0);
    if (IS_ERR (ret)) return ret;
    policy_mark_skip (ctx); } break;
    default : return DROP_UNKNOWN_CT; }
    hairpin_flow |= ct_state->loopback;
    if (!from_l7lb && redirect_to_proxy
    (verdict, ct_status)) { proxy_port =
```

```
(__u16) verdict; send_trace_notify
(ctx, TRACE_TO_PROXY, SECLABEL, 0,
bpf_ntohs (proxy_port), 0,
trace.reason, trace.monitor); return
ctx_redirect_to_proxy6 (ctx, tuple,
proxy_port, false); } if
(!revalidate_data (ctx, &data,
&data_end, &ip6)) return
DROP_INVALID; if (is_defined
(ENABLE_ROUTING) || hairpin_flow) {
struct endpoint_info *ep; ep =
lookup_ip6_endpoint (ip6); if (ep) {
#ifdef ENABLE_ROUTING if (ep->flags &
ENDPOINT_F_HOST) { #ifdef
HOST_IFINDEX goto to_host; #else
return DROP_HOST_UNREACHABLE; #endif
} #endif /* ENABLE_ROUTING */
policy_clear_mark (ctx); return
ipv6_local_delivery (ctx, ETH_HLEN,
SECLABEL, ep, METRIC_EGRESS,
from_l7lb); } } #if
defined(ENABLE_HOST_FIREWALL) &&
!defined(ENABLE_ROUTING) if (*dst_id
== HOST_ID) { ctx_store_meta (ctx,
CB_FROM_HOST, 0); tail_call_static
(ctx, &POLICY_CALL_MAP, HOST_EP_ID);
return DROP_MISSED_TAIL_CALL; }
#endif /* ENABLE_HOST_FIREWALL &&
!ENABLE_ROUTING */ #ifdef TUNNEL_MODE
# ifdef ENABLE_WIREGUARD if
(!dst_remote_ep) # endif /*
ENABLE_WIREGUARD */ { struct
endpoint_key key = {} ; union v6addr
*daddr = (union v6addr *)
&ip6->daddr; key.ip6.p1 = daddr->p1;
key.ip6.p2 = daddr->p2; key.ip6.p3 =
daddr->p3; key.family =
ENDPOINT_KEY_IPV6; ret =
encap_and_redirect_lxc (ctx,
tunnel_endpoint, encrypt_key, & key,
SECLABEL, & trace); if (ret ==
IPSEC_ENDPOINT) goto
encrypt_to_stack; else if (ret !=
DROP_NO_TUNNEL_ENDPOINT) return ret;
} #endif if (is_defined
(ENABLE_HOST_ROUTING)) return
redirect_direct_v6 (ctx, ETH_HLEN,
ip6); goto pass_to_stack; #ifdef
ENABLE_ROUTING to_host : if
(is_defined (ENABLE_HOST_FIREWALL) &&
*dst_id == HOST_ID) {
send_trace_notify (ctx,
TRACE_TO_HOST, SECLABEL, HOST_ID, 0,
HOST_IFINDEX, trace.reason,
trace.monitor); return ctx_redirect
(ctx, HOST_IFINDEX, BPF_F_INGRESS); }
#endif pass_to_stack : #ifdef
ENABLE_ROUTING ret = ipv6_l3 (ctx,
ETH_HLEN, NULL, (__u8 *) &
router_mac.addr, METRIC_EGRESS); if
(unlikely (ret != CTX_ACT_OK)) return
ret; #endif if (ipv6_store_flowlabel
(ctx, ETH_HLEN, SECLABEL_NB) < 0)
return DROP_WRITE_ERROR; #ifdef
ENABLE_WIREGUARD if (dst_remote_ep)
set_encrypt_mark (ctx); else #elif
!defined(TUNNEL_MODE) # ifdef
ENABLE_IPSEC if (encrypt_key &&
tunnel_endpoint) {
set_encrypt_key_mark (ctx,
encrypt_key); # ifdef IP_POOLS
set_encrypt_dip (ctx,
tunnel_endpoint); # endif /* IP_POOLS
*/ # ifdef ENABLE_IDENTITY_MARK
set_identity_mark (ctx, SECLABEL); #
endif /* ENABLE_IDENTITY_MARK */ }
else # endif /* ENABLE_IPSEC */
#endif /* ENABLE_WIREGUARD */ {
#ifdef ENABLE_IDENTITY_MARK ctx->mark
|= MARK_MAGIC_IDENTITY;
set_identity_mark (ctx, SECLABEL);
#endif } #ifdef TUNNEL_MODE
encrypt_to_stack : #endif
send_trace_notify (ctx,
TRACE_TO_STACK, SECLABEL, *dst_id, 0,
0, trace.reason, trace.monitor);
cilium_dbg_capture (ctx,
DBG_CAPTURE_DELIVERY, 0); return
CTX_ACT_OK; }
```

F Error analysis

In our evaluations, several key observations have been identified:

1. The models specifically fail on long codes due to the max length constraint of LLMs as well as information overload from multiple code components.

2. Model outputs are verbose (line-by-line) and do not reflect human annotations, which are intuitive explanations of the source code. This indicated domain adaptation for kernel-based codes (e.g., eBPF) is an important problem to address.

3. The outputs observed from clustering are good but need improvement. We found the tendency of the models to refer and compare to the one-shot example, even when specifically requested in the prompt to generate the summaries independently.

G Chain-of-thought prompting

Chain-of-though (CoT) prompt is a technique where a model is prompted to generate step-by-step explanations of a query. Then, generate an answer based on the prompt as well as the explanations as a context. Due to the paucity of step-by-step explanations in code summarization, we explore CoT under zero-shot settings. Here, we prompt the models to generate a granular/line-by-line summary followed by a paragraph with a concise summary. We observe that the models are not following the instructions, specifically for longer codes. To accommodate longer responses due to granular explanations, we set max_new_tokens to 2048, compared to 256 of non-CoT-based prompts. Since we are only interested in the concise summary, we extract the paragraph starting after "*Concise summary*" in the output. Since the models generate different patterns to create concise summary paragraphs, we manually inspect the outputs by searching "concise" and including all the patterns. In case such a paragraph is absent from the output due to any reason, we keep the whole output. In Mistral-OpenOrca-7B, we achieved a BLEU score of 3.5 (compared to 4.1 on standard zero-shot), ROUGE-1, ROUGE-2, and ROUGE-L of 0.2092, 0.0341, and 0.1816 (compared to 0.2240, 0.0359 and 0.2008 in standard zero-shot), indicating a decline in output quality due to the CoT method. Specifically, we found that the model is hallucinating in some instances (repeating the same line, printing information from instruction-tuning data), which contributes to lowering the quality of outputs. In WizardCoder-15B, we observed way too many patterns for the concise paragraph to effectively extract the concise summary. Although the CoT-based method could generate better summaries when prompted with similar examples with step-by-step summaries, to the best of our knowledge, these types of datasets are unavailable for code summarization, making them not so useful in our setup.

	Rater 1	**Rater 2**	**Rater 3**	**Rater 4**
Rater 1	1	0.8991	0.7392	0.4711
Rater 2	*0.8857*	1	0.9449	0.7806
Rater 3	*0.5768*	*0.6983*	1	0.8611
Rater 4	*0.3479*	*0.6957*	*0.7084*	1

Table 6: Pairwise inter-rater agreement. The lower-triangular matrix represents the Spearman Rank Coefficient (italics), whereas the upper-triangular matrix represents the Pearson Correlation Coefficient;

H Inter-rater Agreement

To evaluate the consistency of the ratings assigned by different raters, we calculated the pairwise inter-rater agreement using two statistical methods: the Pearson Correlation Coefficient and the Spearman Rank Coefficient (Spearman, 1904). The Pearson Correlation Coefficient assesses the linear relationship between the ratings, while the Spearman Rank Coefficient evaluates the monotonic relationship between the rank orders assigned by the raters. Table 6 presents the results of these analyses.
The analysis of the inter-rater agreement using both Pearson and Spearman coefficients provides a com-

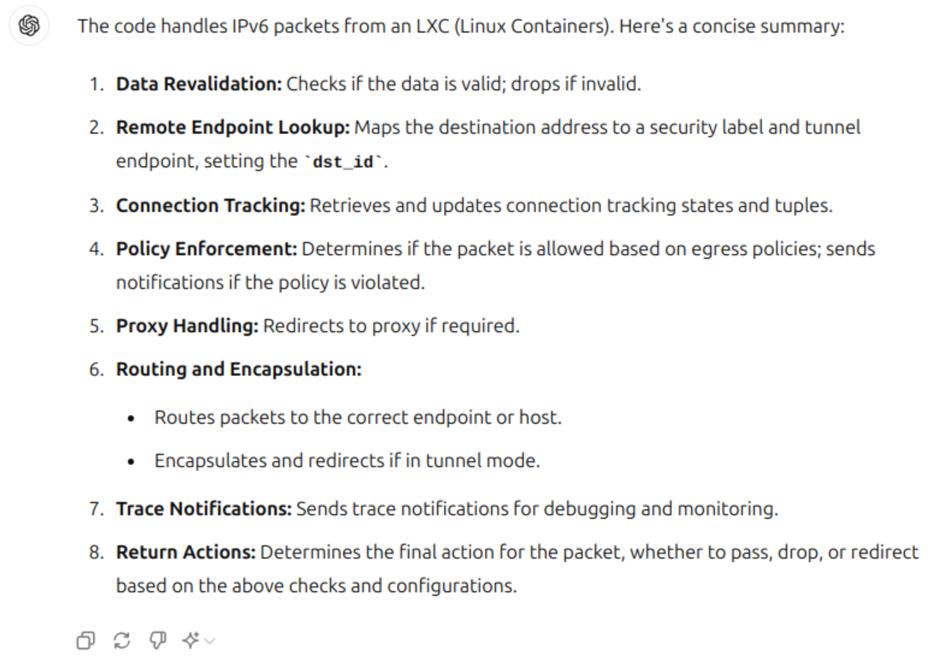
prehensive understanding of the consistency among raters. Given that there were four raters, lower correlation values in some pairwise comparisons are expected due to the increased variability in individual assessments. The generally high Pearson Correlation Coefficients suggest that the raters largely agree in their linear assessments. Meanwhile, the Spearman Rank Coefficients highlight that the raters also exhibit a reasonable degree of agreement in their rank orders, though this agreement is less pronounced in some cases. We calculate the Interclass Correlation Coefficient (ICC) of the ratings. Specifically, we consider ICC3, which is suitable when we have a fixed set of raters for the common examples. We obtain an ICC3 value of 0.750, indicating a moderate to good reliability (Koo and Li, 2016).

I Application of our method in proprietary LLM

Proprietary LLMs (such as ChatGPT) are known to have better-quality outputs than our studied models. Although a full-scale comparison is out-of-scope of the paper, we compared the results of Zero-shot vs. our proposed method on a few examples with GPT-4o. For zero-shot, we prompt the model with: *Generate a short and concise summary of the following code: {Current Code}*. For our proposed one-shot method, we prompt the model with *Code: {Retrieved Code}\n Summary: {Retrieved Summary}\n Now, generate a short and concise summary of the following as per the style of previous example. Code: {Current code}*. We observe consistent improvement in quality when the model is prompted with retrieved example (Figure 3). Specifically, we found that the zero-shot outputs are too verbose, even after prompting to generate a short and concise summary.

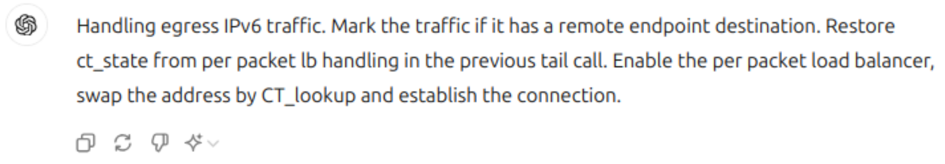
J Evaluation Metrics and Strategy

Details of the models are provided in Appendix C, and the prompt template is presented in Appendix D. To evaluate the generation of eBPF code explanations, we design a unified prompt to enable effective dual-task performance. We employed widely-used metrics, including BLEU (Papineni et al., 2002), ROUGE-L (Lin, 2004), and BERTScore (Zhang et al., 2019). Furthermore, we report the ROUGE score, as it is the most reliable lexical metric for summarization. Additionally, we conduct a human evaluation to assess the relevance of the generated summaries to the input code and their grammatical correctness.


The code handles IPv6 packets from an LXC (Linux Containers). Here's a concise summary:

1. **Data Revalidation:** Checks if the data is valid; drops if invalid.
2. **Remote Endpoint Lookup:** Maps the destination address to a security label and tunnel endpoint, setting the `dst_id`.
3. **Connection Tracking:** Retrieves and updates connection tracking states and tuples.
4. **Policy Enforcement:** Determines if the packet is allowed based on egress policies; sends notifications if the policy is violated.
5. **Proxy Handling:** Redirects to proxy if required.
6. **Routing and Encapsulation:**
   - Routes packets to the correct endpoint or host.
   - Encapsulates and redirects if in tunnel mode.
7. **Trace Notifications:** Sends trace notifications for debugging and monitoring.
8. **Return Actions:** Determines the final action for the packet, whether to pass, drop, or redirect based on the above checks and configurations.

(a) Zero-shot


Handling egress IPv6 traffic. Mark the traffic if it has a remote endpoint destination. Restore ct_state from per packet lb handling in the previous tail call. Enable the per packet load balancer, swap the address by CT_lookup and establish the connection.

(b) Ours

Figure 3: Example of output when the zero-shot prompt is used to summarize using GPT-4o vs. one-shot with our proposed retrieval method.