ProConSuL: Project Context for Code Summarization with LLMs

Vadim Lomshakov[1,*], **Andrey Podivilov**[1,*], **Sergey Savin**[2],
Oleg Baryshnikov[3], **Alena Lisevych**[4], **Sergey Nikolenko**[5,1]
[1]St. Petersburg Department of the Steklov Institute of Mathematics, Russia;
[2]St. Petersburg State University, St. Petersburg, Russia;
[3]HSE University, St. Petersburg, Russia; [4]Independent Researcher;
[5]ITMO University, St. Petersburg, Russia
[*]Equal contribution. **Correspondence:** vadim.lomshakov@gmail.com, sergey@logic.pdmi.ras.ru

Abstract

We propose Project Context for Code Summarization with LLMs (ProConSuL), a new framework to provide a large language model (LLM) with precise information about the code structure from program analysis methods such as a compiler or IDE language services and use task decomposition derived from the code structure. ProConSuL builds a call graph to provide the context from callees and uses a two-phase training method (SFT + preference alignment) to train the model to use the project context. We also provide a new evaluation benchmark for C/C++ functions and a set of proxy metrics. Experimental results demonstrate that ProConSuL allows to significantly improve code summaries and reduce the number of hallucinations compared to the base CodeLlama-7B-instruct model. We make our code and dataset available at `https://github.com/trinity4ai/ProConSuL`.

1 Introduction

State of the art large language models (LLMs) such as GPT-4 (OpenAI, 2023), Claude 3 Opus (Anthropic, 2024b), Claude 3.5 Sonnet (Anthropic, 2024a), and *deepSeek-coder-v2* (Zhu et al., 2024) can code better than ever, exhibiting expert level capabilities in both writing code and comprehending software projects. However, they still suffer from hallucinations and may make wrong conclusions due to the lack of extended context of the entire software project. This is true for people, too: programmers need the project context to understand what a given function does, and the top-down comprehension model (understanding source code from domain context) is noisier than bottom-up comprehension (understanding code statement by statement) because it is hard for the developers to control matching current context with their domain knowledge (Siegmund et al., 2014; Letovsky, 1987). A straightforward solution would be to feed the entire project into the LLM, but it adds a lot of unnecessary information that makes the LLM's job harder and demands extra resources and special tricks to alleviate the quadratic complexity of self-attention (Gemini, 2024; Liu et al., 2023a).

To reduce the amount of information needed to feed an LLM, one has to find out which parts of the project context are crucial for a given practical task such as code summarization. In this work, we propose the **Pro**ject **Con**text for Code **Su**mmarization with **L**LMs framework (ProConSuL) that constructs precise and efficient project-level context for code summarization via formal analysis and adapts LLMs to this context. We focus on C/C++, a programming language very important in practice but severely underrepresented in ML research.

Specifically, we: (1) study different kinds of useful project context for function-level code summarization; (2) represent project context for code summarization based on code structure provided by formal analysis methods and develop a fine-tuning framework for LLMs with supervised fine-tuning on synthetic data and preference alignment; (3) introduce a new real life practical benchmark with automatic and semi-automatic proxy metrics for fast evaluation of code summarization quality for C/C++ and labeling instructions; (4) provide open-sourced training datasets, evaluation benchmark and source code for reproducing our results[1].

The rest of the paper is organized as follows: Section 2 surveys related work, Section 3 introduces the method, including the ProConSuL framework and dataset preparation techniques, Section 4 outlines the evaluation benchmarks and our experimental results, and Section 5 concludes the paper.

2 Related work

Task decomposition for LLMs. This work was partly inspired by Wu et al. (2021b) who intro-

[1]`https://github.com/trinity4ai/ProConSuL`

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 866–880
November 12-16, 2024

duced recursive book summarization, suggesting to decompose a complex task into smaller parts and then compose them back. Importantly, this kind of task decomposition allowed to scale human feedback without requiring the labelers to read the whole book. In the coding domain, Zelikman et al. (2023) suggested to decompose algorithmic tasks into hierarchical natural language function descriptions and then search over combinations of possible function implementations using tests.

Using the formal structure of source code in LLMs. Most previous works in this direction tried to encode code structure information into sequence-based models; e.g., GraphCodeBERT (Guo et al., 2021) uses the data flow extracted from code to help pretrain a BERT-like model, while Wu et al. (2021a) introduce a structure-induced Transformer that applies regularization to the self-attention mechanism by masking the attention matrix with adjacency matrices of different graph representations. Another direction of research models source code with graph neural networks (GNN) (Allamanis et al., 2017; Zhang et al., 2022). Hellendoorn et al. (2020) suggest to combine self-attention layers with GNN layers, bridging the gap between global attention in Transformers and inherently local GNNs that rely on message passing. To extend the context to large software projects, Ma et al. (2024) use an agent that traverses a project's graph representation and collects information necessary to solve a specific task. In this work, we in turn use formal representations of code for task decomposition and augmenting the model context.

Extending the context. The problem might be solved if we were able to provide full context of a large programming project to an LLM. This, however, runs into the quadratic complexity of self-attention. There are several approaches to alleviate this quadratic complexity, including sparse attention mechanisms (Beltagy et al., 2020; Child et al., 2019; Zaheer et al., 2020), low-rank decomposition for the matrix of self-attention weights (Choromanski et al., 2020; Wang et al., 2020), or chunking attention to constrain quadratic complexity to small subsets of the input, either with a recurrent architecture (Bulatov et al., 2022; Guo et al., 2023; Hua et al., 2022; Ma et al., 2023), fitting more tokens by interpolating positional embeddings (Chen et al., 2023), or with other tricks such as hashing (Kitaev et al., 2020) or blockwise attention (Liu et al., 2023a). However, large context sizes are still challenging for LLMs to use efficiently (Liu et al., 2023b), and a better solution would choose the contents of this long context wisely.

Evaluation metrics for text generation. Most common evaluation metrics compare generated text to a reference, including n-gram-based metrics and embedding-based metrics such as ROUGE (Lin, 2004), METEOR (Banerjee and Lavie, 2005), or BERTScore (Zhang et al., 2019); the problem here is the lack of high quality reference texts. One idea is to do away with them completely and evaluate based on the source document instead: He et al. (2008) thus arrive at the ROUGE-C metric, but in our case it is inapplicable because natural language summaries and code represent different modalities. Liu et al. (2023c) and Zheng et al. (2023) use a different strong LLM to evaluate text generation automatically. Another approach is to evaluate different properties of text separately; e.g., Deutsch et al. (2021) estimate the quality of a summary by a set of question-answer pairs automatically generated from the reference. We also note that large language models are notorious for hallucinating, i.e., introducing erroneous facts in their output, and detecting hallucinations is also an important problem (Fadeeva et al., 2024; Manakul et al., 2023).

3 Method

3.1 Context-augmented Code Summarization

To achieve state of the art code summarization in a natural language, we propose the **Pro**ject **Con**text for Code **Su**mmarization with **L**LMs framework (PROCONSUL) that provides a large language model with precise information about the code structure provided by program analysis methods such as a compiler or IDE language services and uses task decomposition derived from the code structure (in our case, the call graph).

PROCONSUL consists of four major components: (1) it *builds a call graph* to provide the context from callees and uses a special instruction format that includes a new context section, including the list of callee names and code summary pairs (see Appendix A, Table 7); (2) it *synthesizes a training dataset* by using a reference model and applying special filtering, as detailed in Section 3.2; (3) it performs *supervised fine-tuning* and *preference alignment* to adapt the model to use project context; (4) at inference time, it performs *recursive summarization* to propagate facts along the project call graph. Specifically, recursive summarization (a) constructs the call graph and contracts loops,

(b) traverses the resulting tree in topological order, summarizing each function with summaries of its callees as context, and (c) for a loop, generate summaries in some order and do another iteration with new summaries as context.

We focused on C/C++ as the target programming language because it remains an important and very popular language in industry but is underrepresented in AI research: most recent papers and benchmarks cover Python and/or Java. At the same time, C/C++ is more complex for program analysis.

To obtain the call graph, we use a Clang-based tool as the industry standard for parsing C/C++. For all experiments, we use instruction-tuned versions of models from the CodeLlama family with 7B and 34B parameters (Rozière et al., 2024).

Supervised fine-tuning (SFT). Preliminary experiments with in-context learning for the base CodeLlama models led only to quality degradation with increased project context. Therefore, we used SFT to adapt the model to a new prompt distribution (see Section 3.2). We trained rank-stabilized LoRA (rsLoRA) adapters (Kalajdzievski, 2023) with hyperparameters following Biderman et al. (2024): LR=3.e-5, LoRa Rank=16, LoRA Modules='all', constant scheduler, 8 bit quantization, efficient batch size 512; for training, we used 2 NVIDIA V100 16Gb GPUs.

Alignment. Preference alignment is a great fit for our task because it helps to align model behaviour with desired outputs while using a relatively small amount of high quality feedback (Ouyang et al., 2022). The goal here would be to decrease the likelihood of verbose and trivial code summaries and increase the likelihood of correct and concise code summaries at the same time. We used the odds ratio preference optimization algorithm (ORPO) (Hong et al., 2024), a modification of direct preference optimization (DPO) (Rafailov et al., 2023) that combines SFT and DPO into a single phase; ORPO is an easy to implement and more computationally efficient counterpart of known RL methods such as proximal policy optimization (PPO) (Schulman et al., 2017).

For the data, we generated 950 positive examples with GPT-4o by using prompt engineering and the callee's project context (see Appendix A). We filtered the functions and generated the callee's context for every function by vanilla CodeLlama-7B, and then collected several negative examples for each positive code summary; negatives were generated by other versions of CodeLlama that suffered from hallucinations, verbosity, triviality, or factual mistakes. The final training set for ORPO contains 3000 negative-positive pairs. We trained LoRA adapters with LR=1e-4, LoRa rank=16, Modules='all', linear scheduler, 8 bit quantization, efficient batch size 32, ORPO beta=0.1; for training, we used 2 NVIDIA V100 16Gb GPUs. We use the same instruction template with callee context as during the SFT phase (see Appendix A).

3.2 SFT dataset preparation

First, we extracted and ranked the most popular GitHub repositories written in C/C++ with GitHub Public Repository Metadata[2], filtering projects where we were able to automatically generate the JSON compilation database with project compilation commands for Clang[3]. In total, we selected 25 projects of different sizes, including *linux*, *redis*, *llvm-project*, *curl*, and others. Then we ran a Clang-based tool to build the global call graph and extract all function declarations from the project together with their metadata including callee–caller relations, docstrings (if they exist) etc. For the test set, we separately selected 5 repositories from domains that are similar to our enterprise codebase and exclude them from train: *ffmpeg*, *openssl*, *wrk*, *llvm/clang/tidy*, and *libuv* (see also Section 4). To prevent contamination and data leaks, we removed all (near) duplicates between test and train sets.

To filter and generate synthetic docstrings, we used the same CodeLlama-instruct model as we had used for fine-tuning; prompts used for generation are shown in Appendix A. We applied a custom crafted set of filters before and after generating the summaries, including (but not limited to; see the repository for more details): (1) remove samples with very short or very long code, leaving function bodies between 50 and 4000 symbols; (2) remove function declarations without bodies or functions with an empty body; (3) include functions with comments only in English; (4) remove synthetic code summaries with stop words such as "fixme", "deprecated", and others; (5) remove trivial and verbose synthetic code summaries according to automated metrics (see Section 4); (6) remove very shot or long synthetic code summaries (under 2 and over 70 words); (7) remove synthetic code summaries if they contain code entities (expressions, statements

[2]https://www.kaggle.com/datasets/pelmers/github-repository-metadata-with-5-stars/versions/8

[3]https://clang.llvm.org/

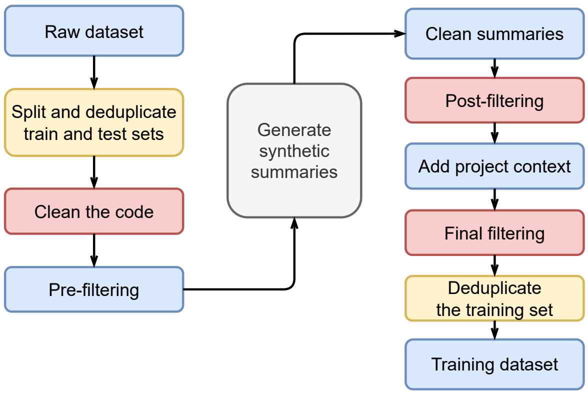

Figure 1: Training dataset preparation pipeline

or code blocks) from the function body. The data preparation pipeline is illustrated in Figure 1. After filtering, we obfuscated callee names (replaced the name with a random string) with probability 0.5 to force the model to use context information, and used the instruction prompt with callee context shown in Appendix A to get the final training set.

3.3 Useful project context categories

Before performing experiments, we studied various useful categories of the project context and their importance for code summarization. For this purpose, we sampled 50 random functions from popular C/C++ repositories (according to the number of stars, forks, and watchers); two researchers of our team manually labeled these functions as follows. Each annotator wrote a code summary using only the function body, then corrected it using IDE, documentation, and Web search, and finally noted project context categories that helped them to comprehend the source code.

As a result, we extracted seven code context categories that are possibly useful from the human perspective and collected statistics on what category is most popular and promising for future research. Table 1 shows their descriptions and percentage of occurrences. We see that most popular context category (46% of the cases) is "Callees", functions that are called from the target function. In 22% of the cases, there is no need for any context to summarize the source code: naming is enough or the code is self-explanatory. Moreover, some categories such as "Web search" or "Readme" may become unnecessary if the model has enough domain knowledge. Thus, we find that the most important category is "Callees", with second and third places occupied by "Usages" and "Classes".

Context category	Description	%
Callees	Information about callees: code, docstring, code summary, filename etc.	46%
Classes	Information about the struct or class: documentation, code summary, source code	22%
Usages	Information about callers: call site context, docstring, code summary, code, function name etc.	20%
Web search	Meanings of abbreviations, documentation, usage examples, new knowledge	20%
File	Information from the source file where the function is located: other functions, file-level docstring, filename, classes	18%
Readme	Information from the *readme.md* file or project documentation: domain info about the project, formatting information etc.	10%
Globals	Information about a global variable: declaration, code summary, docstring	8%
No context	There is no need for context, the function body contains enough information	22%

Table 1: Project context categories. Percentages do not sum to one because multiple categories can apply to the same function

4 Evaluation Benchmark and Results

4.1 Motivation

To the best of our knowledge, there is no suitable publicly available benchmark for evaluating C/C++ code summarization models. For example, CodeXGLUE (Husain et al., 2019) includes code summarization but does not cover C/C++ and uses a reference-based metric BLEU; this is problematic since reference texts are usually the original docstrings that are very noisy and often contain information that cannot be derived from function body and project context (Mu et al., 2023). Muennighoff et al. (2024) suggest to use backtranslation and use the Pass@K metric, but do not control the style and other metrics and are sub-optimal for our case of code summarization because a short summary cannot contain enough information to generate back a long function body. Therefore, in this work we design and implement a new benchmark and metrics for C/C++ code summarization.

4.2 Evaluation criteria and proxy metrics

Criteria. We begin by formulating the criteria that a good code summary should meet in the form of labeling instructions; we have aligned these criteria with real software developers and designed a corresponding evaluation system. We distinguish two groups of criteria: style-related and content-related. The criteria are shown in Table 2; verbosity and

Criterion	Description	Proxy metric
Verbosity	A summary is verbose if it contains redundant information or an overly detailed description (e.g., a description of local variables), explains the function statement by statement with no added value (e.g., "calling function `foo` with argument `x`), or contains repeated information	An automatic proxy metric based on the length of summary and repetition of substrings in the summary
Triviality	A summary is trivial if all the information it contains can be deduced from the function signature: name, argument names and types, return type	Overlapping words between function signature and summary
Sufficiency	A summary is sufficient if it contains enough information to understand what the function actually does without looking at the function body, and implementation details are included if they are crucial to use the function correctly	*Sufficiency via QA*: questions and binary answers (manually) prepared in the test set, GPT-4 used as the QA model; *Sufficiency via GPT*: GPT-4 is asked to compare two summaries and return which one is better
Factual correctness	A summary is factually correct if it does not contain facts or details that can be proven wrong based on the information given to the LLM (e.g., "if $x > 0$ the function returns `true`" while the actual condition is "$x < 0$")	Use GPT-4 to check correctness, double-checking its output against the list of possible mistakes and hallucinations
Hallucinations	A summary contains hallucinations if it contains information that cannot be inferred from the repository or general knowledge, e.g., mentions nonexisting code entities (variables, functions etc.), invariants or guarantees that cannot be inferred (thread-safety, time/memory complexity etc.), or additional claims about implicit behaviours, usage, or meta-knowledge (e.g., saying that "`normalize_frame()` is called for every frame of a video" while no context about its usage has been provided)	Use GPT-4 to check for hallucinations, double-checking its output against the list of possible mistakes and hallucinations
Random facts	A summary contains random facts if it includes claims that do not help understand the code and seem out of place (e.g., "C is a popular yet complicated language")	No proxy metric developed; Section 4.3 shows that random facts are almost never generated

Table 2: Evaluation criteria for code summaries (all prompts are given in Appendix A)

triviality are style-related and the rest are content-related. For all criteria except *Sufficiency* we ask the labelers to provide a binary 0/1 score; for *Sufficiency*, we perform a side-by-side comparison of two summaries. Full labeling instructions are provided in the github repository.

Proxy metrics. Manual annotation is expensive and time consuming, especially for the C/C++ programming language, where this process might take hours. While we emphasize that human opinion is still the gold standard for final evaluation and cannot be fully replaced by automated metrics, to streamline hypothesis testing and perform, e.g., validation set experiments we have designed proxy metrics for each criterion from the annotation guide. These metrics are also detailed in Table 2. Using these proxy metrics significantly speeds up manual annotation. We have evaluated proxy metrics for agreement with human annotators against answers generated by other modifications of CodeLlama (base version, various SFT and SFT+ORPO versions), using them to generate code summaries on the test set. For the triviality criterion, we labeled 150 points (25 for each of 6 models) by 3 annotators, with every annotations labeled by two human assessors. The proxy metric agreed with human annotation in 136 cases, with precision 0.769 and recall 0.714. For verbosity, our proxy metric was very useful during early experiments but stopped working for the best models because the few remaining verbose summaries copied source code rather than just repeated themselves. For sufficiency, we sampled 25 pairs of summaries and labeled it by two human assessors. GPT-4 agrees with humans 18 times out of 25 with 7 ties, while the QA-based metric agreed with humans only 8 times with 3 wrong answers and 14 ties. As a result, we find that top-level LLMs such as GPT-4 can serve as excellent proxy metrics for code summary evaluation (but they are harder to scale and cannot be used for closed codebases, i.e., basically for any enterprise solution) while the QA-based metric is worse even though it is also GPT-based.

Test data. We selected 5 repositories (*ffmpeg*, *openssl*, *wrk*, *llvm/clang/tidy*, *libuv*) for testing, excluded them from training, and sampled 25 functions, 5 per repository, filtering out third-party functions and function-like macros. We used this microbenchmark for manual evaluation of our models. For hyperparameter tuning and model selection, we sampled 20 separate points for the validation set (we use functions from *libuv* and *ffmpeg* whose

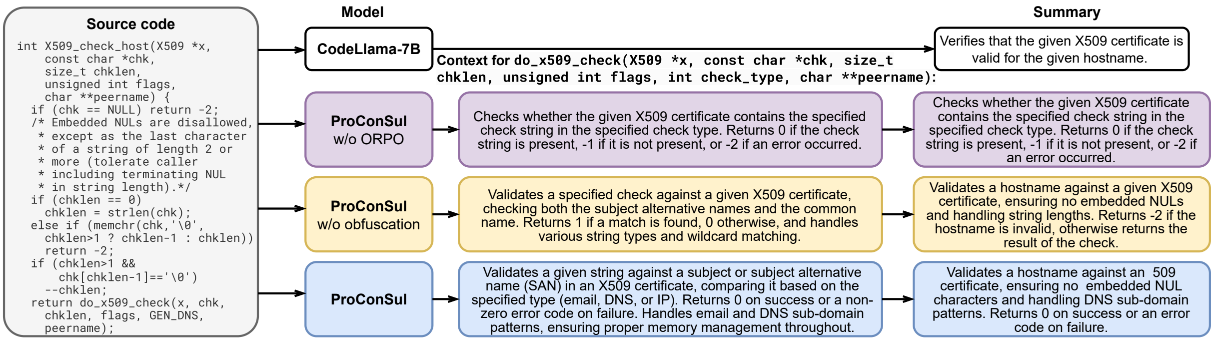

Figure 2: A comparison of sample annotations for a hostname validation function

Model		SFT	ORPO	Synth. data	Obfuscation	Sufficiency (win/lose/tie vs. PROCONSUL)	Verbosity ↓	Triviality ↓	Fact. correctness ↑	Hallucinations ↓	Random facts ↓
CodeLlama-7B-instruct						1/20/4	1	9	22	**1**	**0**
PROCONSUL		✓	✓	✓	✓		2	**0**	**23**	**1**	**0**
		✓		✓	✓	1/22/2	**0**	5	17	3	1
		✓				0/23/2	**0**	14	21	4	**0**
			✓			5/4/16	6	**0**	21	2	**0**
		✓	✓	✓		3/6/16	4	0	21	7	**0**
CodeLlama-34B-instruct						2/14/9	15	5	14	9	7
PROCONSUL-34B			✓				**3**	**1**	**22**	**1**	**0**

Table 3: Experimental results (out of 25 annotations).

callee graphs do not intersect with the test set).

4.3 Evaluation Results

Experiments. For our main experimental results, 3 annotators have performed manual labeling over 25 annotations each, with every annotations labeled by two human assessors. Table 3 shows the results of PROCONSUL compared to the baseline of CodeLlama-7B-instruct and a number of variations that comprise an ablation study for different parts of our approach. It is clear that PROCONSUL outperforms the baseline and variations with different parts of the approach switched off. We have also tested ORPO on the larger 34B model (Table 3) and obtained significantly improved results.

For the ablation study, we note that fine-tuning on original docstrings usually only hurts the model; filtering can also lead to very short or trivial docstrings. The SFT phase with synthetic data lets the model improve style and make summaries less verbose and less trivial; it has also adapted PROCONSUL to having context in the prompt. RLAIF also improves the model, and in our case it allowed to

Model	Sufficiency (win/lose/tie vs. PROCONSUL)	Verbosity ↓	Triviality ↓	Fact. correctness ↑	Hallucinations ↓	Random facts ↓
PROCONSUL-7B		**2**	**0**	**23**	1	**0**
GPT-4o	2/5/18	3	**0**	22	**0**	**0**
GPT-4o with callee context	7/3/15	**2**	1	22	**0**	**0**

Table 4: Comparison with GPT-4o.

Model	Sufficiency (win/lose/tie vs. PROCONSUL)	Verbosity ↓	Triviality ↓	Fact. correctness ↑	Hallucinations ↓	Random facts ↓
PROCONSUL-7B		**2**	**0**	**23**	1	**0**
PROCONSUL-7B without context	1/6/18	**3**	**0**	**23**	1	**0**

Table 5: Comparing PROCONSUL with and without callee context.

improve summaries further than we could achieve by prompting alone, even with a small training set (1000 positive examples for using context). The "Random facts" criterion, included from our prior experience, has proven to be almost unnecessary: modern LLMs do not add random facts (except for rare cases when they generate code).

Comparison with state of the art closed LLMs. Table 4 shows a comparison of PROCONSUL and GPT-4o with and without callee context. The results are comparable and differ only in a few cases. We have also tried to apply our recursive inference approach to prompt GPT-4o, and it has improved the GPT-4o's sufficiency metric, as expected. We also note that our approach allows to efficiently collect information from the callee graph into a limited context; for some data points in the *llvm* project, the size of the callee graph reaches nearly 30000 functions.

We have also conducted an experiment to analyze the importance of having this context, evaluating the results of a model without callee context. Table 5 shows that without the context, the model significantly loses in the sufficiency metric. This supports our hypotheses that (1) context is used by the model and (2) it actively improves the sufficiency metric, i.e., it is not merely a distillation of GPT-4o.

Qualitative evaluation. Figure 2 and Appendix B show a selection of characteristic sample summaries for the CodeLlama-7B-Instruct baseline and different versions of PROCONSUL. Note that different versions of PROCONSUL produce different context summaries for the same function.

From this qualitative analysis we conclude that: (1) it is hard to achieve non-trivial and non-verbose summaries via pure prompt engineering, the baseline generates either trivial one-sentence docstrings or very verbose responses, and zero-shot context does not help (Ex. 1–5, Appendix B); (2) SFT without ORPO and synthetic data on original docstrings (Ex. 6–9) often leads to trivial answers; adding synthetic data without ORPO improves style but produces more trivial and less correct summaries; (3) using ORPO without SFT leads to more verbose summaries; (4) removing obfuscation increases hallucinations; also, in Fig. 2 we see how without obfuscation the model just summarizes the function body while the full PROCONSUL adds important information from context; (5) improvements extend to the 34B version of the models as well (Ex. 10).

Real-world applications. This work has arisen out of a real world project on AI for code. Real world applications include, for instance, generating docstrings on a private codebase that has insufficient documentation. Importantly, the expert acceptance rate for our results is high (with a large difference between vanilla and trained models), so results of this work are already being used in a production environment. Another application is generating synthetic data for code generation models. We have tested fine-tuning on our synthetic summaries, and the pass@1 metric has increased compared to the model trained on original docstrings.

5 Conclusion

In this work, we have introduced the PROCONSUL framework that gathers and uses project-level context for code summarization in C/C++, including a new method for synthetic data collection and labeling, a method for fine-tuning LLMs via a combination of SFT and preference alignment, and a new benchmark based on real world C/C++ functions. We show that a proper use of the project context allows to significantly improve code summaries and reduce the number of hallucinations by using precise information from the context. We hope that this research is a stepping stone to bridging the gap between formal source code analysis and LLMs.

6 Limitations

The main practical limitation here is that the evaluation dataset in this work is rather small, restricting the robustness of our results. Unfortunately, scaling the evaluation much further would be beyond our capacity since factual correction and hallucination metrics are not fully automated and require human supervision, which in the case of code summaries is slow, requires high expertise, and is therefore expensive. The evaluations shown in this work have been performed by our research and development team, with two human assessors labeling every summary.

Acknowledgements

The work of Sergey Nikolenko shown in Sections 3 and 4 has been supported by the Russian Science Foundation grant no. 22-11-00135, https://rscf.ru/en/project/22-11-00135/.

References

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *CoRR*, abs/1711.00740.

Anthropic. 2024a. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

Anthropic. 2024b. Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *CoRR*, abs/2004.05150.

Dan Biderman, Jose Gonzalez Ortiz, Jacob Portes, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, Cody Blakeney, and John P. Cunningham. 2024. Lora learns less and forgets less. *Preprint*, arXiv:2405.09673.

Aydar Bulatov, Yuri Kuratov, and Mikhail S. Burtsev. 2022. Recurrent memory transformer. *Preprint*, arXiv:2207.06881.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *Preprint*, arXiv:2306.15595.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy J. Colwell, and Adrian Weller. 2020. Rethinking attention with performers. *CoRR*, abs/2009.14794.

Daniel Deutsch, Tania Bedrax-Weiss, and Dan Roth. 2021. Towards question-answering as an automatic metric for evaluating the content quality of a summary. *Transactions of the Association for Computational Linguistics*, 9:774–789.

Ekaterina Fadeeva, Aleksandr Rubashevskii, Artem Shelmanov, Sergey Petrakov, Haonan Li, Hamdy Mubarak, Evgenii Tsymbalov, Gleb Kuzmin, Alexander Panchenko, Timothy Baldwin, Preslav Nakov, and Maxim Panov. 2024. Fact-checking the output of large language models via token-level uncertainty quantification. *Preprint*, arXiv:2403.04696.

Team Gemini. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *Preprint*, arXiv:2403.05530.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphcodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. *Preprint*, arXiv:2306.14893.

Tingting He, Jinguang Chen, Liang Ma, Zhuoming Gui, Fang Li, Wei Shao, and Qian Wang. 2008. Rouge-c: A fully automated evaluation method for multi-document summarization. In *2008 IEEE International Conference on Granular Computing*, pages 269–274.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Jiwoo Hong, Noah Lee, and James Thorne. 2024. Orpo: Monolithic preference optimization without reference model. *Preprint*, arXiv:2403.07691.

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc V. Le. 2022. Transformer quality in linear time. *Preprint*, arXiv:2202.10447.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Damjan Kalajdzievski. 2023. A rank stabilization scaling factor for fine-tuning with lora. *Preprint*, arXiv:2312.03732.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *CoRR*, abs/2001.04451.

Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325–339.

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.

Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023a. Ring attention with blockwise transformers for near-infinite context. *Preprint*, arXiv:2310.01889.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023b. Lost in the middle: How language models use long contexts. *Preprint*, arXiv:2307.03172.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023c. G-eval: Nlg evaluation using gpt-4 with better human alignment. *Preprint*, arXiv:2303.16634.

Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Luke Zettlemoyer. 2023. Mega: Moving average equipped gated attention. *Preprint*, arXiv:2209.10655.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? *Preprint*, arXiv:2406.01422.

Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. 2023. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *Preprint*, arXiv:2303.08896.

Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2023. Developer-intent driven code comment generation. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 768–780. IEEE Press.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. *Preprint*, arXiv:2308.07124.

OpenAI. 2023. GPT-4 technical report. *Preprint*, arXiv:2303.08774.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, volume 36, pages 53728–53741. Curran Associates, Inc.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 378–389, New York, NY, USA. Association for Computing Machinery.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. *CoRR*, abs/2006.04768.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021a. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1078–1090, Online. Association for Computational Linguistics.

Jeff Wu, Long Ouyang, Daniel M. Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul F. Christiano. 2021b. Recursively summarizing books with human feedback. *CoRR*, abs/2109.10862.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big bird: Transformers for longer sequences. *CoRR*, abs/2007.14062.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. 2023. Parsel: Algorithmic reasoning with language models by composing decompositions. *Preprint*, arXiv:2212.10561.

Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to represent programs with heterogeneous graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 378–389, 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. *CoRR*, abs/1904.09675.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Preprint*, arXiv:2306.05685.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Prompt template:
`[INST] <<SYS>>`
Write a docstring for the following C++ code. You should write the shortest possible docstring: no code, only one sentence.
<</SYS>>
{code}
[/INST] Here is a revised version of the docstring with improved style (no code):

Example:
`[INST] <<SYS>>`
Write a docstring for the following C++ code. You should write the shortest possible docstring: no code, only one sentence.
<</SYS>>

```
static inline int wnv1_get_code(GetBitContext *gb,
                int shift, int base_value)
{
    int v = get_vlc2(gb, code_vlc, CODE_VLC_BITS, 1);
    if (v == 8)
        return get_bits(gb, 8 - shift) << shift;
    else
        return base_value + v * (1 << shift);
}
```

`[/INST]`
Here is a revised version of the docstring with improved style (no code):

Table 6: Prompt template used to generate the synthetic dataset, examples.

A Prompt structure

In this section we show the prompt templates and provide examples. In particular:

- Table 6 contains the prompt template used to generate synthetic data for fine-tuning;
- Table 7 shows the prompt template for summarization that we used for both training and inference;
- Table 8 shows the prompt template for a strong LLM (GPT-4o) used to compute the pairwise GPT-based proxy metric for sufficiency;
- Table 9 provides the prompt template for finding factual mistakes and hallucinations with a strong closed LLM (GPT-4o);
- Table 10 shows the prompt template for a strong closed LLM (GPT-4o) used to compute the QA-based proxy metric for sufficiency.
- Table 11 shows the prompt template for a strong closed LLM (GPT-4o) used to generate positive examples for alignment dataset.

Prompt template:
`[INST] <<SYS>>`
You are an expert in Programming. Below we have two sections separated by four hyphens: "—-".
The second section is a C++/C code snippet.
Above that code snippet we have additional info to help you out: a list of function callees with their docstrings (separated by an asterisk).
Return a line of summary that describes the function.
`<</SYS>>`

```
{name1}({params1}): {doc1}
*
{name2}({params2}): {doc2}
*
...
----
{code} [/INST]
```

Example:
`[INST] <<SYS>>`
You are an expert in Programming. Below we have two sections separated by four hyphens: "—-".
The second section is a C++/C code snippet.
Above that code snippet we have an additional info to help you out: a list of function callees with their docstrings (separated by an asterisk).
Return a line of summary that describes the function.

```
<</SYS>>
get_vlc2(GetBitContext *s, const VLCElem *table,
         int bits, int max_depth): Retrieves a
         variable-length code from the given
         GetBitContext using the specified VLCElem
         table and maximum depth.
*
get_bits(GetBitContext *s, int n): Returns the next
         n bits from GetBitContext, where n is a
         positive integer less than or equal to 25.
----
static inline int wnv1_get_code(GetBitContext *gb,
        int shift, int base_value)
{
    int v = get_vlc2(gb, code_vlc, CODE_VLC_BITS, 1);
    if (v == 8)
        return get_bits(gb, 8 - shift) << shift;
    else
        return base_value + v * (1 << shift);
} [/INST]
```

Table 7: Prompt template used at training and inference time, examples.

B Examples

The long table provides several characteristic examples of specific C/C++ functions from our evaluation set and summaries produced by different methods; the qualitative results and conclusions we derive from these example are discussed in Section 4.3.

System prompt:
Below you have a code snippet with 2 summaries delimited with summary_A and summary_B tags.
Please tell which one of them is more comprehensive and complete,
i.e. covers more crucial aspects of the code and gives a clearer description of what the function does,
or if they are equally comprehensive. Please be as concise as possible, I don't have much time.
User prompt:

```
<code>
{code} </code>

<summary_A>
{doc1} </summary_A>

<summary_B>
{doc2} </summary_B>
```

Which one is more complete? Are they comparable?
Your answer: [Model Answer]
User prompt:

Based on your thoughts give a final answer. Return a single character: "A" for summary_A, "B" for summary_B and "C" if they are comparable.
Your response (one letter): [Model Answer]

Table 8: Prompt template used to obtain the pairwise sufficiency score.

System prompt:
You are a knowledgeable C/C++ code expert. You are here to help your colleagues with abstractive code summarization task. Your answers should be concise and substantial. Follow your instructions strictly. Try to give your answers in the form of a short list. Your colleagues would appreciate it if you give a short and accurate answer.
User prompt:
Below we have a C/C++ code of a function and a docstring for that function (delimited with XML tags). We need to decide whether this function docstring gives a factual high-level summary of the code. Patiently go over each statement from this function docstring. Then give a list of details this docstring gets wrong - if it makes a mistake and says something that is not true - tell us; start by providing a short quotation. Also, mention if the docstring contains hallucinations - statements that can not be extracted from the given code or general context; give an explanation. Recall that the purpose of this docstring is a high-level summarization, so don't expect a comprehensive code summary. If the docstring omits details, it is fine, it is not a mistake or disadvantage from our perspective, do not mention it in your review. Answer template example:
Wrong details:
- ...
Statements from the docstring that can not be extracted from the given code or general context:
- ...

<code>
{code} </code>
<docstring>
{doc} </docstring>

Table 9: Prompt template used to get factual mistakes and hallucinations.

System prompt:
You are an expert in Programming. You are here to help your colleagues with abstractive code summarization task. For your help to be effective you need to follow given instructions strictly. Your task is to answer Yes or No to every question using only the information in the provided docstring. You should use the provided docstring as the only source of truth. Give a separate answer to every question in order. One answer per question on separate lines. Answer only Yes or No. If you are unsure about the answer to a question, add a comment to your answer on the same line.
User prompt:
Docstring:
"""{doc}"""
Questions:
Does the docstring mention claim1?
Does the docstring mention claim2?
...

Table 10: Prompt template used for QA-based Sufficiency metric.

System prompt:
You are a knowledgeable C/C++ code expert. Your task is to assist me with an abstractive code summarization task. I need you to provide an example of a well-thought-out, comprehensive, yet concise function summary. Write 1 to 4 short sentences that summarize the function below. If applicable, describe the purpose and effects of the code, but omit unnecessary details. Focus on abstraction and highlighting key points: be as terse as possible, like a Terminator.
User prompt:
Below is some information gathered from our code repository.
To help you create a comprehensive summary, we provide additional context: a list of callees with their docstrings.
This might help you understand the broader context of our project.
The structure is as follows: a list of callees with docstrings, followed by the function code, and then a blank line for your response.
The function and the list of callees are delimited with XML tags for clarity (<code> and <callees list> respectively).

```
<callees list>
{name1}({params1}): {doc1}
*
{name2}({params2}): {doc2}
*
...</callees list>

<code>
{code} </code>
```

Write a concise function summary below (only 1-4 sentences, as if you are a Terminator):

Table 11: Prompt template used for positive examples generation.

Code	Context and summaries

Example 1:
`int X509_check_host(X509 *x,`
`  const char *chk, size_t chklen,`
`  unsigned int flags, char **peername) {`
`  if (chk == NULL) return -2;`
`  /* Embedded NULs are disallowed, except as`
`   * the last character of a string of length`
`   * 2 or more (tolerate caller including`
`   * terminating NUL in string length).*/`
`  if (chklen == 0)`
`    chklen = strlen(chk);`
`  else if (memchr(chk,'\0', chklen>1 ?`
`    chklen - 1 : chklen)) return -2;`
`  if (chklen>1 && chk[chklen-1]=='\0')`
`    --chklen;`
`  return do_x509_check(x, chk, chklen,`
`    flags, GEN_DNS, peername);`
`}`
	Baseline summary:
	Verifies that the given X509 certificate is valid for the given hostname.
	PROCONSUL Context:
	`do_x509_check(X509 *x, const char *chk, size_t chklen, unsigned int flags, int check_type, char **peername)`: Validates a given string against a subject or subject alternative name (SAN) in an X509 certificate, comparing it based on the specified type (email, DNS, or IP). Returns 0 on success or a non-zero error code on failure. Handles email and DNS sub-domain patterns, ensuring proper memory management throughout.
	PROCONSUL summary:
	Validates a hostname against an X509 certificate, ensuring no embedded NUL characters and handling DNS sub-domain patterns. Returns 0 on success or an error code on failure.

Example 2:
`int uv_udp_try_send(uv_udp_t* handle,`
`  const uv_buf_t bufs[],`
`  unsigned int nbufs,`
`  const struct sockaddr* addr) {`
`  int addrlen;`
`  addrlen = uv__udp_check_before_send(`
`    handle, addr);`
`  if (addrlen < 0)`
`    return addrlen;`
`  return uv__udp_try_send(`
`    handle, bufs, nbufs, addr, addrlen);`
`}`
	Baseline summary:
	Sends data over a UDP socket.
	PROCONSUL Context:
	`uv__udp_check_before_send (uv_udp_t* handle, const struct sockaddr* addr)`: Validates and determines the length of a socket address for UDP send operations. Returns appropriate error codes for invalid handle types, connected states, and address families.
	*
	`uv__udp_try_send(uv_udp_t* handle, const uv_buf_t bufs[], unsigned int nbufs, const struct sockaddr* addr, unsigned int addrlen)`: Attempts to send UDP data without blocking. Returns an error if already sending or if the send operation fails due to a non-retryable error. Initializes the message structure and handles deferred binding if necessary.
	PROCONSUL summary:
	Attempts to send UDP data without blocking by validating the socket address and initializing the send operation. Returns an error code if the send operation fails or if the handle is already sending.

Example 3:
`static void write_index(AVFormatContext *s)`
`{`
`  AVIOContext *pb = s->pb;`
`  WtvContext *wctx = s->priv_data;`
`  int i;`
`  write_chunk_header2(s,`
`    &ff_index_guid, 0x80000000);`
`  avio_wl32(pb, 0);`
`  avio_wl32(pb, 0);`
`  for (i = 0; i < wctx->nb_index; i++) {`
`    WtvChunkEntry *t = wctx->index + i;`
`    ff_put_guid(pb,  t->guid);`
`    avio_wl64(pb, t->pos);`
`    avio_wl32(pb, t->stream_id);`
`    avio_wl32(pb, 0); // checksum?`
`    avio_wl64(pb, t->serial);`
`  }`
`  wctx->nb_index = 0; // reset index`
`  finish_chunk_noindex(s);`
`  if (!wctx->first_index_pos)`
`    wctx->first_index_pos =`
`      wctx->last_chunk_pos;`
`}`
	Baseline summary:
	Writes the index chunk to the WTV file. The index chunk contains a list of all the chunks in the file, along with their positions and serial numbers. The index is reset after this function is called.
	PROCONSUL Context:
	`write_chunk_header2( AVFormatContext *s, const ff_asf_guid *guid, int stream_id)`: Writes a chunk header for an ASF format context, updating the length and storing the previous chunk's position.
	*
	`avio_wl32( AVIOContext *s, unsigned int val)`: Writes a 32-bit unsigned integer to an AVIOContext in little-endian byte order.
	*
	`ff_put_guid( AVIOContext *s, const ff_asf_guid *g)`: Writes a 16-byte GUID to the specified AVIOContext. Ensures the GUID is 16 bytes in size and aborts if the condition is not met.
	*
	`avio_wl64( AVIOContext *s, uint64_t val)`: Writes a 64-bit unsigned integer to an AVIOContext in little-endian format. Breaks the 64-bit value into two 32-bit chunks and writes each chunk separately.
	*
	`finish_chunk_noindex( AVFormatContext *s)`: Finalizes a chunk by updating the chunk length, writing any necessary padding, and incrementing the serial number.
	PROCONSUL summary:
	Finalizes the index chunk by writing all stored chunk entries, resets the index, and updates the chunk header. Ensures the index is written in little-endian format and includes necessary metadata.

Code	Context and summaries

Example 4:

```
void UseAfterMoveFinder::getUsesAndReinits(
    const CFGBlock *Block,
    const ValueDecl *MovedVariable,
    llvm::SmallVectorImpl<
      const DeclRefExpr*> *Uses,
    llvm::SmallPtrSetImpl<
      const Stmt*> *Reinits) {
  llvm::SmallPtrSet<const DeclRefExpr*, 1>
    DeclRefs;
  llvm::SmallPtrSet<const DeclRefExpr*, 1>
    ReinitDeclRefs;
  getDeclRefs(Block, MovedVariable, &DeclRefs);
  getReinits(Block, MovedVariable,
    Reinits, &ReinitDeclRefs);
  // All references to the variable that
  // aren't reinitializations are uses.
  Uses->clear();
  for (const DeclRefExpr *DeclRef : DeclRefs) {
    if (!ReinitDeclRefs.count(DeclRef))
      Uses->push_back(DeclRef);
  }
  // Sort the uses by their
  // occurrence in the source code.
  std::sort(Uses->begin(), Uses->end(),
     [](const DeclRefExpr *D1,
        const DeclRefExpr *D2) {
     return D1->getExprLoc() < D2->getExprLoc();
  });
}
```

Baseline summary:
Finds all uses and reinitializations of a moved variable in a given CFGBlock.

PROCONSUL Context:
No context

PROCONSUL summary:
Identifies and collects all uses and reinitializations of a moved variable in a given CFGBlock. Sorts the uses by their source location.

Example 5:

```
static int aeApiAddEvent(aeEventLoop *eventLoop,
    int fd, int mask) {
  aeApiState *state = eventLoop->apidata;
  /* avoid valgrind warning */
  struct epoll_event ee = {0};
  /* If the fd was already monitored
   * for some event, we need a MOD operation.
   * Otherwise we need an ADD operation. */
  int op =
    eventLoop->events[fd].mask == AE_NONE ?
    EPOLL_CTL_ADD : EPOLL_CTL_MOD;
  ee.events = 0;
  /* Merge old events */
  mask |= eventLoop->events[fd].mask;
  if (mask & AE_READABLE) ee.events |= EPOLLIN;
  if (mask & AE_WRITABLE) ee.events |= EPOLLOUT;
  ee.data.fd = fd;
  if (epoll_ctl(state->epfd,op,fd,&ee) == -1)
    return -1;
  return 0;
}
```

Baseline summary:
Adds or modifies an event in the epoll instance.

PROCONSUL Context:
No context

PROCONSUL summary:
Monitors or modifies file descriptor events in the epoll event loop. Adds or modifies events based on the current state and new mask, and returns 0 on success or -1 on failure.

<table>
<tr><th>Code</th><th>Context and summaries</th></tr>
<tr><td>

Example 6:

```
static void hnm_update_palette(
    AVCodecContext *avctx, const uint8_t *src,
    uint32_t size) {
  Hnm4VideoContext *hnm = avctx->priv_data;
  GetByteContext gb;
  uint8_t start, writeoffset;
  uint16_t count;
  int eight_bit_colors;
  eight_bit_colors =
    src[7] & 0x80 && hnm->version == 0x4a;
  // skip first 8 bytes
  bytestream2_init(&gb, src + 8, size - 8);
  while (bytestream2_tell(&gb) < size - 8) {
    start = bytestream2_get_byte(&gb);
    count = bytestream2_get_byte(&gb);
    if (start == 255 && count == 255)
      break;
    if (count == 0)
      count = 256;
    writeoffset = start;
    while (count > 0) {
      hnm->palette[writeoffset] =
        bytestream2_get_be24(&gb);
      if (!eight_bit_colors)
        hnm->palette[writeoffset] <<= 2;
      hnm->palette[writeoffset] |= (0xFFU << 24);
      count--;
      writeoffset++;
    }
  }
}
```

</td><td>

Baseline summary:
Updates the palette of the HNM4 video context based on the given source data.

ProConSuL Context:
No context.

ProConSuL without synthetic data summary:
update palette from the palette table

ProConSuL summary:
Updates the palette of the Hnm4 video context by reading and decoding palette data from a source buffer. Handles both 8-bit and 16-bit color modes.

</td></tr>
<tr><td>

Example 7:

```
static int add_info(AVIOContext *bc,
  const char *type, const char *value)
{
  put_str(bc, type);
  put_s(bc, -1);
  put_str(bc, value);
  return 1;
}
```

</td><td>

Baseline summary:
Adds information to the AVIOContext.

ProConSuL Context:
No context.

ProConSuL without synthetic data summary:
Add a new info tag to the current tag.

ProConSuL summary:
Encodes and writes type and value to an AVIOContext buffer using specific functions.

</td></tr>
<tr><td>

Example 8:

```
static int pjs_probe(const AVProbeData *p)
{
  char c;
  int64_t start, end;
  const unsigned char *ptr = p->buf;
  if (sscanf(ptr, "%"SCNd64",%"SCNd64",%c",
      &start, &end, &c) == 3) {
    size_t q1pos = strcspn(ptr, "\"");
    size_t q2pos = q1pos +
      strcspn(ptr + q1pos + 1, "\"") + 1;
    if (strcspn(ptr, "\r\n") > q2pos)
      return AVPROBE_SCORE_MAX;
  }
  return 0;
}
```

</td><td>

Baseline summary:
Probes the input buffer for the presence of a valid PJS file header, returning a score indicating the likelihood of success if the header is found.

ProConSuL Context: No context.

ProConSuL without synthetic data summary:
check if the file is a pjs file

ProConSuL summary:
Determines if the provided buffer contains a valid probe data format, returning a score if successful. Parses the buffer to extract start and end positions and a character, then checks for the presence of a newline character.

</td></tr>
<tr><td>

Example 9:

```
size_t zmalloc_size(void *ptr) {
  void *realptr = (char*)ptr-PREFIX_SIZE;
  size_t size = *((size_t*)realptr);
  /* Assume at least that all the
   * allocations are padded at sizeof(long)
   * by the underlying allocator. */
  if (size&(sizeof(long)-1)) size +=
      sizeof(long)-(size&(sizeof(long)-1));
    return size+PREFIX_SIZE;
}
```

</td><td>

Baseline summary:
Returns the size of the memory block pointed to by ‘ptr‘, which must have been allocated by ‘zmalloc‘. The size is stored in the first few bytes of the block, and is assumed to be padded to a multiple of ‘sizeof(long)‘ by the underlying allocator.

ProConSuL Context: No context.

ProConSuL on the original docstring:
Return the size of the allocation pointed to by ’ptr’.

ProConSuL summary:
Calculates the actual memory size of a memory block by retrieving the stored size, adjusting for alignment if necessary, and adding the prefix size.

</td></tr>
</table>

<table>
<tr><th>Code</th><th>Context and summaries</th></tr>
<tr><td>

Example 10: Summarization with the 34B model

```
static int script_wrk_lookup(lua_State *L) {
  struct addrinfo *addrs;
  struct addrinfo hints = {
    .ai_family   = AF_UNSPEC,
    .ai_socktype = SOCK_STREAM
  };
  int rc, index = 1;
  const char *host    = lua_tostring(L, -2);
  const char *service = lua_tostring(L, -1);
  if ((rc = getaddrinfo(
      host, service, &hints, &addrs)) != 0) {
    const char *msg = gai_strerror(rc);
    fprintf(stderr,
      "unable to resolve %s:%s %s\n",
      host, service, msg);
    exit(1);
  }
  lua_newtable(L);
  for (struct addrinfo *addr = addrs;
      addr != NULL; addr = addr->ai_next) {
    script_addr_clone(L, addr);
    lua_rawseti(L, -2, index++);
  }
  freeaddrinfo(addrs);
  return 1;
}
```

</td><td>

Baseline CodeLlama-34B summary:
Looks up the host and service using getaddrinfo and returns a table of addresses.

PROCONSUL PROCONSUL-34B context:
`script_addr_clone(lua_State *L, struct addrinfo *addr)`: Clones a ‘struct addrinfo‘ object in Lua by creating a new userdata, setting its metatable, and copying the source address data. Returns the cloned address.
*
`lua_tostring(L,i)`: Converts the value at index ‘i‘ in the Lua stack to a string, returning the string or ‘NULL‘ if the conversion fails.
*
`lua_newtable(L)`: Creates a new empty table in the Lua stack.

PROCONSUL-34B summary:
Performs a DNS lookup for a given host and service, creating a Lua table of resolved addresses. Clones each address into the table and returns it. Exits the program if the lookup fails.

</td></tr>
</table>