Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)

Toufique Ahmed
University of California, Davis
Davis, California, USA
tfahmed@ucdavis.edu

Kunal Suresh Pai
University of California, Davis
Davis, California, USA
kunpai@ucdavis.edu

Premkumar Devanbu
University of California, Davis
Davis, California, USA
ptdevanbu@ucdavis.edu

Earl T. Barr
University College London & Google Brain
London, UK
e.barr@ucl.ac.uk

ABSTRACT

Large Language Models (LLM) are a new class of computation engines, "programmed" via prompt engineering. Researchers are still learning how to best "program" these LLMs to help developers. We start with the intuition that developers tend to consciously and unconsciously collect semantics facts, from the code, while working. Mostly these are shallow, simple facts arising from a quick read. For a function, such facts might include parameter and local variable names, return expressions, simple pre- and post-conditions, and basic control and data flow, *etc.*

One might assume that the powerful multi-layer architecture of transformer-style LLMs makes them *implicitly* capable of doing this simple level of "code analysis" and extracting such information, while processing code: but are they, really? If they aren't, could *explicitly* adding this information help? Our goal here is to investigate this question, using the *code summarization task* and evaluate whether automatically augmenting an LLM's prompt with semantic facts *explicitly*, actually helps.

Prior work shows that LLM performance on code summarization benefits from embedding a few code & summary exemplars in the prompt, before the code to be summarized. While summarization performance has steadily progressed since the early days, there is still room for improvement: LLM performance on code summarization still lags its performance on natural-language tasks like translation and text summarization.

We find that adding semantic facts to the code in the prompt actually does help! This approach improves performance in several different settings suggested by prior work, including for *three* different Large Language Models. In most cases, we see improvements, as measured by a range of commonly-used metrics; for the PHP language in the challenging CodeSearchNet dataset, this augmentation actually yields performance surpassing 30 BLEU[1]. In addition, we have also found that including semantic facts yields a substantial enhancement in LLMs' line completion performance.

[1]Scores of 30-40 BLEU are considered "Good" to "Understandable" for natural language translation; see https://cloud.google.com/translate/automl/docs/evaluate.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639183

KEYWORDS

LLM, Code Summarization, Program Analysis, Prompt Engineering

ACM Reference Format:
Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639183

1 INTRODUCTION

Large language models (LLMs) often outperform smaller, custom-trained models on tasks, especially when prompted with a "few-shot" set of exemplars. LLMs are pre-trained on a self-supervised (masking or de-noising) task, using vast amounts of data, and exhibit surprising emergent behaviour as training data and parameter counts are scaled up. They excel at many tasks with few-shot (or even zero-shot) learning: with just a few exemplar input-output pairs inserted first in the prompt, the models can generate very good outputs for a given input! Few-shot learning works so well with LLMs that it is unclear whether sufficient task-specific data can ever be gathered to train a customized model to rival their performance [3, 12]. LLMs are ushering in a new era, where prompt engineering, to carefully condition the input to an LLM to tailor its massive, but generic capacity, to specific tasks, will become a new style of programming, placing new demands on software engineers.

We propose *Automatic Semantic Augmentation of Prompts* (*ASAP*), a new method for constructing prompts for software engineering tasks. The *ASAP* method rests on an analogy: an effective prompt for an LLM, for a task, relates to the facts a developer thinks about when manually performing that task. In other words, we hypothesize that prompting an LLM with the syntactic and semantic facts a developer considers when manually performing a task *will* improve LLM performance on that task. To realise this hypothesis, *ASAP* augments prompts with semantic facts automatically extracted from the source code using *semantic code analysis*.

We illustrate the *ASAP* methodology first on code summarization. This task takes code, usually a function, and summarizes it using natural language; such summaries can support code understanding to facilitate requirements traceability and maintenance.

ASAP uses a few-shot prompting because its effectiveness. *ASAP* finds relevant shots using *BM25*, the current state of the art in finding few-shot exemplars that are "semantically close" to the target function [48], in our case, the function-to-summarize, by querying the LLM's training data. When instantiating *ASAP* for the summarization task, we equipped it to extract the following semantic facts: the repository name, the fully qualified name of the name of the target function, its signature, the AST tags of its identifiers, and its data flow graph (Section 3.4). These facts are presented to the LLM as separate, labelled, fields[2]. The model is then provided with the function-to-summarize, exemplars (along with facts extracted from each), and asked to emit a summary. We confirm our hypothesis that augmenting prompts with semantic facts can improve LLM performance on the code completion task. We evaluated *ASAP*'s benefits on the high-quality (carefully de-duplicated, multi-project) CodeSearchNet [32] dataset.

In summary, we find that in *all cases*, our approach of automatic semantic augmentation *improves average performance* on *several commonly-used metrics*. For almost all languages,the average improvement comfortably surpasses the 2-BLEU threshold noted by Roy *et al.* [57], below which BLEU results are unreliable predictors of human preference. For Go, gains are still significant, and just slightly less than 2; for PHP, we see an improvement of *4.6 BLEU*, reaching a SOTA high-point of 32.73 on the well-curated, de-duplicated, CodeSearchNet dataset.

Our principal contributions follow:

- The *ASAP* approach for software engineering tasks using facts derived from code.
- We evaluate *ASAP* on the code summarization task on the code-davinci-002, text-davinci-003. and GPT-3.5-turbo models against a few-shot prompting baseline built using vanilla *BM25* (Section 4.1).
- We find that the *ASAP* approach statistically significantly improves LLM performance on the code summarization task. In almost all cases, we observe statistically significant improvements of almost, or in excess of, 2 BLEU; and, for PHP, we break 30 BLEU for the first time (to our knowledge) on this challenging dataset.
- We find that *ASAP* also leads to improved performance on the code-completion task.

All the data, evaluation scripts, and code needed to reproduce this work will be available at https://doi.org/10.5281/zenodo.7779196, and can be reproduced on any available language models. Our experiments suggest that *ASAP* works well with any language model powerful enough to leverage few-shot prompting.

2 BACKGROUND & MOTIVATION

Large Language Models (LLM) are a transformative technology: they are essentially a new kind of computation engine, requiring a new form of programming, called prompt engineering. We first contextualise *ASAP*, our contribution to prompt engineering. Finally, we discuss code summarization as a sample problem to demonstrate *ASAP*'s effectiveness.

[2]A full example is rather long, and is included in the repository due to paper length limitations.

2.1 Few-shot Learning in Software Engineering

LLMs are now widely used in Software Engineering for many different problems: code generation [14, 34], testing [38, 42], mutation generation [10], program repair [18, 35, 36, 48], incident management [6], and even code summarization [3]. Clearly, tools built on top of pre-trained LLM are advancing the state of the art. Beyond their raw performance at many tasks, two key factors govern the growing dominance of pretrained LLM, both centered on cost. First, training one's own large model, or even extensively fine-tuning a pre-trained LLM, requires expensive hardware. Second, generating a supervised dataset for many important software engineering tasks is difficult and time-consuming, often beyond the sources of all but the largest organizations.

In contrast to overall LLM trends, there are some smaller models, specialized for code, that have gained popularity, *e.g.*, Polycoder [67] or Codegen [49]. Despite these counterpoints, we focus on LLM rather than small models, because, while small models can be fine-tuned, they don't do very well at few-shotting, and thus are not helpful when only small amounts of data are available. The few-shot approach is key because it brings into reach many problems, like code summarization, for which collecting sufficient, high-quality, project- or domain-specific training data to train even small models from scratch is challenging.

With few-shot learning, the actual model parameters remain unchanged. Instead, we present a few problem instances along with solutions (*i.e.*, problem-solution pairs as "the exemplars") to a model and ask it to complete the answer for the last instance ("the test input"), for which we do not provide a solution. Thus with each *exemplar* consisting of an $\langle input, output \rangle$ pair, and just a test-input $input_t$ (without the corresponding, desired $output_t$), the final prompt looks like:

$$prompt \leftarrow exemplar_1 \;||\; exemplar_2 \;||\; exemplar_3 \;||\; input_t$$

With this prompt, the LLM generates $output_t$, mimicking the input-output behavior illustrated by the exemplars in the prompt. In practice, this approach performs quite well.

When it works, few-shotting allows us to automate even purely manual problems, since generating a few exemplar samples is relatively easy. In this paper, we experiment with the code-davinci-002 model. We discuss models in more detail in Section 3.2.

2.2 Prompting LLMs to Reason

Human Reasoning involves using evidence, logical thinking, and arguments to make judgments or arrive at conclusions [31, 51]. Natural language processing (NLP) researchers have developed approaches to reason about specific scenarios and improve performance. Approaches like "Chain of thought" [66] and "step-by-step" [40] require generating intermediate results ("lemmas") and utilizing them in the task at hand. Such approaches appear to work on simpler problems like school math problems even without providing them with "lemmas" , because, for these problems, models are powerful enough to generate their own "lemmas"; in some cases just adding "let's think step by step" seems sufficient (Kojima et al. [40]).

We tried an enhanced version of the "step-by-step" prompt, with few-shots, on code summarization. We found that the model underperformed (getting about 20.25 BLEU), lower even than our vanilla BM25 baseline (24.97 BLEU). With zero-shot Kojima-style "step by step" prompt, the models perform even worse. To induce the model to generate steps, and finally a summary, we framed the problem as chain of thought, *and* included few-shot samples containing both intermediate steps ("lemmas") and final comments. The reasoning is that, on the (usually challenging) code-related tasks, models need to explicitly be given intermediate "lemmas", *derived from code*, to be able to reason effectively about most software engineering tasks, which tend to be more complex and varied than school maths.

Fortunately, mature tools for code analysis are available. We can readily derive "lemmas", *viz.*, analysis products, using code analysis tools, rather than expecting the models to (perhaps implicitly) derive them, during on-task performance. We directly embed analysis products into the prompt we give the language model, and evaluate the benefits of such analysis products. The information we derive and add are based on our own intuitions about the kinds of "lemmas" that developers consciously or unconsciously consider as they seek to understand and summarize code.

We find that providing such information improves LLM performance. We remind the reader that most work involving large language models (LLMs) usually uses some form of prompt engineering to boost performance. In this paper, we show that the $\mathcal{ASAP}$ approach, which augments prompts with code analysis products, improves on previous prompting approaches.

2.3 Summarizing Code

Well-documented code is much easier to maintain; thus, experienced developers usually add, *e.g.*, function summary headers. However, summary comments may become outdated, as projects evolve [11, 22]. Automated code summarization is thus a well-motivated task, which has attracted a great deal of attention; and considerable progress (albeit incremental, over many years) has been made. Initially, template-based approaches were popular [17, 26, 27, 56, 61]; however, creating a list of templates with good coverage is very challenging. Later, researchers focused on the retrieval-based (IR) approach [17, 26, 27, 56], where existing code (with a summary) is retrieved based on similarity-based metrics. However, this promising approach only worked if a similar code-comment pair could be found in the available pool.

Meanwhile, the similarity of code summarization to Neural Machine Translation (NMT), (one can think of generating an English summary of code as producing a representation of "the same meaning in a different language") led to research that adapted Neural Machine Translation (NMT) to code summarization. Numerous studies have been conducted in this area [1, 30, 33, 41]. Some have combined previous approaches, such as template-based and retrieval-based approaches, using neural models [69], and have reported promising results. Such neural methods for NLP have vastly improved, due to the Transformer architectural style.

Until recently, pre-trained language models such as CodeBERT, CodeT5, and CodeT5+ performed best for code summarization.

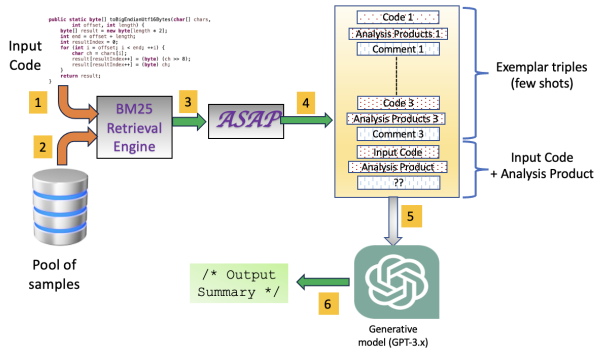

Figure 1: Different steps of $\mathcal{ASAP}$. (1) Input code and (2) Pool of samples are given the BM25 engine, which matches the given input code against the pool and (3) retrieves best-matching samples, *viz.* 3 input+output pairs. These examples are processed by $\mathcal{ASAP}$ to produce a prompt (4) including 3 exemplars. Each exemplar includes a function definition, the results of analyzing that definition, and its associated comment; the input code is finally appended, along with its analysis product. Exemplar details are in Figure 2. The final prompt is sent via API call (5) to the GPT-3.x model; the returned output, *e.g.*, summary (6) is returned by GPT-3x.

However, Large Language Models (LLMs) now typically outperform smaller pre-trained models on many problems. Ahmed & Devanbu [3] report that LLMs can outperform pre-trained language models with a simple prompt consisting of just a few samples already in the same project; this work illustrates the promise of careful construction of prompt structures (*c.f.* "prompt engineering"). We present $\mathcal{ASAP}$ here as another general principle of prompt engineering. We emphasize, again, that progress in code summarization (and other applications of AI to SE, such as code patching, defect detection, testing *etc*) has been incremental, as in the field of NMT, where practical, usable translation systems took decades to emerge. Thus *incremental advances are still needed, and helpful,* and we contribute our work to this long-term enterprise.

3 DATASET & METHODOLOGY

We now discuss our dataset, models, and methodology.

3.1 Dataset

Our experiments use the widely used CodeSearchNet [32] dataset; CodeSearchNet was constructed by extracting the first paragraph of the function prefix documentation, subject to some restrictions (*e.g.* length). It is a carefully de-duplicated, multi-project dataset, which allows (more demanding) cross-project testing. De-duplication is key: Code duplication in machine learning models can deceptively inflate performance metrics a lot, when compared to de-duplicated datasets [7, 46, 59].

It is part of the CodeXGLUE [47] benchmark, which comprises 14 datasets for 10 software engineering tasks. Many models have been evaluated on this dataset. CodeSearchNet contains thousands of samples from six different programming languages (*i.e.*, Java, Python, JavaScript, Ruby, Go, PHP). However, we did not use the entire test dataset, which would have been prohibitively expensive

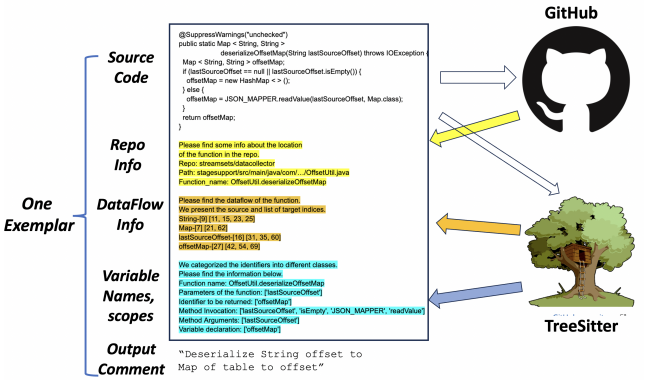

Figure 2: Components of an *ASAP* Exemplar. Source Code and Output Comment are extracted from the retrieved pool sample. The Repo info is derived from the source code using GitHub; the Dataflow Info and tagged Identifiers with labels is obtained from an analysis using Treesitter.

Language	#of Training Samples	#of Test Samples
Java	164,923	1000
Python	251,820	1000
Ruby	24,927	1000
JavaScript	58,025	1000
Go	167,288	1000
PHP	241,241	1000

Table 1: Number of training and test samples.

and slow using ours models API endpoints; instead, we selected 1000 samples[3] uniformly at random from each language. Since the original dataset is cross-project and we sampled it uniformly, our subsample includes cross-project data. In addition, we subsetted this dataset for same-project few-shotting, following Ahmed and Devanbu [3]: we sort same-project data by creation date (using `git blame`). Now, we use the temporal order to make sure that only temporally *earlier* samples are used the few-shot exemplars; this is realistic, since only older, already existing data is available for use. We will delve deeper into this same-project dataset in Section 4.3.

As mentioned earlier, we don't use any parameter-changing training on the model; we just insert a few exemplars selected from the training subset into the few-shot prompt. Table 1 lists the count of training & test samples used in our experiments.

3.2 The Models

In earlier work, transformer-based pre-trained language models offered significant gains, in both NLP and software engineering. Pre-trained language models can be divided into three categories: encoder-only, encoder-decoder, and decoder-only models. While encoder-decoder models have initially shown success on many tasks, decoder-only LLMs are now more scaleable and effective for numerous tasks.

Encoder-Decoder model. BERT is one of the earliest pre-trained language models [15]; it was pre-trained on two self-supervised tasks: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). Later, RoBERTa [45] was introduced with some minor modifications to BERT. Using only MLM training, it outperforms BERT. CodeBERT [21] and GraphCodeBERT [25] introduced these ideas to Software Engineering. Although CodeBERT and GraphCodeBERT are encoder-only models, they can be applied to code summarization after fine-tuning, cascaded to a decoder trained during fine-tuning. Ahmed & Devanbu report that polyglot models, which are fine-tuned with multilingual data, outperform their monolingual counterparts [4]. They also report that identifiers play a critical role in code summarization tasks. PLBART [2], CodeT5 [64], and CodeT5+ [63] also include pre-trained *decoders* and are reported to work well for code summarization tasks. More recently, very large scale (decoder-only) auto-regressive LLMs (with 175B+ parameters) have been found to be successful at code summarization with few-shot learning, without any explicit training. In the next section, we will briefly introduce the three OpenAI models we considered for our experiments.

Decoder-only model. In generative pre-training, the task is to auto-regressively predict the next token given the previous tokens moving from earlier to later. This unidirectional auto-regressive training prevents the model from pooling information from future tokens. The newer generative models such as GPT [52], GPT-2 [53] and GPT-3 [12], are also trained in this way, but they have more parameters, and are trained on much larger datasets. Current large language models, such as GPT-3, have around (or more than) 175B parameters. These powerful models perform so well, with few-shot prompting, that interest on task-specific parameter-adjustment via fine-tuning has reduced.

Codex is a GPT-3 variant, intensively trained on code and natural language comments. The Codex family consists of two versions: Codex-Cushman, which is smaller, with 12B parameters, and Codex-Davinci, the largest, with 175B parameters. The Codex model is widely used, for various tasks. Our experiments mostly target the Code-Davinci model, particularly Code-Davinci-002, which excels at translating natural language to code [14] and supports code completion as well as code insertion[4]. Some new variants, Text-Davinci-003 & GPT-3.5-turbo, are also available; unlike the Codex variants, these models understand and generate both natural language and code. Although optimized for chat, GPT-3.5-turbo also performs well on traditional completion tasks. Text-Davinci-003 is a completion model like Code-Davinci-002. We study how our prompt enhancement works using the Text-Davinci-003 & GPT-3.5-turbo models.

3.3 Retrieving Exemplars from Training Data

As noted earlier, few-shot learning works quite well, when used with very large models. We prompt the model with a small number of ⟨*problem*, *solution*⟩ exemplars, and ask it to solve a new problem. However, carefully selecting exemplars for few-shot learning is helpful. Nashid *et al.* discovered that retrieval-based exemplar selection is helpful for problems such as assertion generation and program repair [48]. Following their recommendation, we use the *BM25* IR algorithm to select relevant few-shot exemplars from the training set. *BM25* [55] is a frequency-based retrieval method which improves upon TF-IDF [54]. We noted a substantial improvement

[3]Please see experimental power discussion in Section 7.

[4]https://openai.com/

over the same fixed exemplars in few-shot learning, as detailed in Section 4.1. Nashid et al. compare several retrieval methods, and found *BM25* works best; we therefore use it, as well.

3.4 Automatic Semantic Prompt Augmentation

This section presents the three semantic facts we selected to enhance $\mathcal{ASAP}$'s prompts and the $\mathcal{ASAP}$ pipeline (See Figure 2). The choice of these facts comes from applying our central hypothesis, *viz.* that augmenting prompts with what developers think about when working on a task, to the code summarization task. $\mathcal{ASAP}$ is not tied to any specific semantic facts or static analysis; it can easily incorporate others, as discussed later.

Repository Name & Path. Augmenting prompts with domain-specific information can improve LLM performance on various tasks. Prior work suggests that augmenting prompts with *code* from the *same repository* improves performance in code generation tasks [60]. We argue that basic repository-level *meta*-information, such as the repository name and the complete path to the repository, provides additional context. For example, repository names like "tony19/logback–android", "apache/parquet–mr", and "ngageoint/geo–package–android" all connect a function to a specific domain (*e.g.*, android, apache, geo-location), which can enhance the understanding of the target code to be summarized. Figure 2 (yellow part) presents an example of how we enhance the prompt with repository-level information. Similar to the repository name, the path to the function can also contribute to the model.

Tagged Identifiers. Prior work suggests that language models find more value in identifiers, rather than code structure, when generating code summaries [4]. However, identifiers play different roles in code. Local variables, function names, parameters, global variables *etc.*, play different parts in the functioning of the method in which they occur; a developer reading the code is certainly aware of the roles of identifier, simply by identifying the scope and use. Thus, augmenting prompts with the *specific roles* of identifiers could help the model better "understand" the function. We use tree-sitter to traverse the function's AST and gather identifiers, along with their roles. Figure 2 (blue part) presents a sample example showing how we enhanced the prompt of the function with tagged identifiers. Although the model has access to the token sequence of the code, and thus also all the identifiers, them to the model in a tagged form might a) save the model some compute effort, and b) better condition the model's output.

Data Flow Graph (DFG). Guo *et al.* introduced the GraphcodeBERT model, which uses data flow graphs (DFG) instead of syntactic-level structures like abstract syntax trees (ASTs) in the pre-training stage [25]. GraphcodeBERT outperformed CodeBERT [21] on various software engineering (SE) tasks. We incorporate this DFG information into the few-shot exemplars; we conjecture that this provides the model a better semantic understanding of each exemplar, and the target example. Figure 2 (orange) presents a sample showing the Data Flow Graph (DFG) we used for our experiments. Each line contains an identifier with its index and the index of the identifiers to which that particular data flows. Unlike repo and tagged identifiers, the data flow graph can be very long, making it inconvenient to add the complete data flow to the prompt. In the case of long prompts, we only kept the first 30 lines of the DFG in the prompt. In addition to identifiers, the DFG also provides a better understanding of the importance of identifiers in the function.

Use Case & Completion Pipeline. $\mathcal{ASAP}$ has 3 components: an LLM, a pool of available exemplars (labeled input-output pairs, *e.g.*, code with comments), and a static analysis tool for deriving facts from code (See Figures 1 and 2).

A configuration file specifies these components. Once configured, a developer invokes $\mathcal{ASAP}$ on a function body C_{in} (Figure 1), for which an output (*e.g.*,, code summary) is desired. $\mathcal{ASAP}$ uses C_{in} as a BM25 query over the its sample pool to get a result set of exemplar candidates $ec_1, ec_2, \ldots$, where each ec_i is a pair of the form $\langle input_i, output_i \rangle$; in our context, $input_i$ is the function definition and $output_i$ is the function header comment. BM25 chooses the $input_i$s that match best with the given C_{in}. $\mathcal{ASAP}$ then applies program analyses to both the input C_{in} and the several exemplar inputs $input_i$s, yielding analysis products ap_{in} and several ap_is.

Each exemplar e_i (Figure 2) is the triple: $\langle input_i, ap_i, output_i \rangle$, where each triple illustrates, for the LLM, how input source code $input_i$ relates, *via* the analysis product ap_i, to the output $output_i$. The final prompt is then "$e_1 \,||\, e_2 \,||\, e_3 \,||\, C_{in} \,||\, ap_{in}$". $\mathcal{ASAP}$ queries an LLM with that prompt, and returns the completion (*e.g.*, natural language summary).

By default, $\mathcal{ASAP}$ is configured with analyses to extract repository info, tag identifiers, construct DFGs. These analyses are independent and are their outputs are separately labeled in the prompt. For example, Figure 2 shows the output of the DFG analysis in $\mathcal{ASAP}$'s constructed prompt. These few shot examples, are augmented and inserted into the prompt: the code, repository info, tagged identifiers, the DFG, and the desired (Gold) summary are all included in each few-shot. The target example includes just analysis product, and the LLM is prompted to produce the desired output.

In prior work using "chain of thought" [66] or "step by step" [40] reasoning, no such information is given to the model; instead, the prompt simply helps it organize its reasoning about the sample into a sequence of instructions. Here, rather than having the model do its own reasoning, we shape its reasoning externally by using simple program analyses, since we can get very precise information from very efficient analysis tools. Each few-shot example includes source code, derived information, and conclusion (summary), thus providing exemplary "chains of thought" for the model to implicitly use when generating the desired target summary. Figure 1 presents the overall pipeline of our approach that we apply to each sample. The BM25 engine matches input code against a sample pool, $\mathcal{ASAP}$ processes resulting examples to create a prompt, and the final prompt is sent to the GPT-3.x model via API, yielding a summary as output.

Next, we describe how we evaluate this pipeline.

3.5 Metrics

BLEU [50] is the most widely-used, similarity-based measure for code summarization [57] and commit log generation [16]. BLEU counts the *fraction* of n-grams (usually for $n \in [1..4]$), that occur in both generated candidates and one or more reference translations; the geometric mean of these fractions is the BLEU, usually normalized to the range 0-100. At *sentence* granularity, BLEU tends to overly penalize candidate translations when few (or none) of the

longer n-grams co-occur, so "Sentence BLEU" has been criticized for correlating poorly with human judgment. Various smoothing techniques [13, 23, 44] have been used, to reduce Sentence BLEU's sensitivity to sparse *n*-gram matches, and better align it with human quality assessment. We report data on two variants: BLEU-CN, which uses a kind of Laplacian smoothing [2, 3, 8, 21, 33, 47, 64] and BLEU-DC, which uses newer smoothing methods [29, 65]. Other proposed metrics such as BERTScore [28, 70], BLEURT [58], NUBIA [37], are computationally expensive, not widely used and thus not readily comparable with prior work for benchmarking.

Given all these options, metrics for code summarization and, independently, for commit-log generation [16], have been debated [24, 28, 57]. In this paper, we follow prior work and primarily use BLEU-CN; this facilitates the comparison of our results with prior work. The CodeXGLUE benchmark recommends BLEU-CN, and most newer models [3, 21, 64] use this metric. We, however, *have not neglected other measures.* Besides BLEU-CN, and BLEU-DC, we also report results using ROUGE-L [43] and METEOR [9].

In all cases, $\mathcal{A}SAP$ achieves significant overall improvements: we observe gains greater than 2.0 BLEU for all programming languages except for Go (Table 3). We contend that gains greater than 2.0 BLEU are important for two reasons. Roy *et al.* [57] provide arguments, grounded on human subject study that *for code summarization* (our central task), that a gain of 2.0 or more BLEU is more likely to correspond with human perception of improvement. Second, we argue that **even smaller gains matter** (especially if repeatable and statistically significant) since incremental progress on such tasks accumulates, towards strong practical impact, as evidenced by decades-long work in natural language translation.

In addition to code summarization, we evaluated $\mathcal{A}SAP$ approach on the code completion task. The standard metrics used for this task are *exact match* (did the completion match exactly) and *edit similarity* (how close is the completion to the expected sequence). Here, too, $\mathcal{A}SAP$ achieves significant overall improvements.

3.6 Experimental Setup & Evaluation Criteria

Our primary model is OpenAI's code-davinci-002. We use the beta version, via its web service API. To balance computational constraints like rate limits and our desire for robust estimates of performance, we chose to use 1000 samples[5] per experimental treatment (one treatment for each language, each few-shot selection approach, with $\mathcal{A}SAP$, without $\mathcal{A}SAP$ *etc.*).

Our experiments yielded statistically significant, interpretable results in most cases. Each 1000-sample trial still took 5 to 8 hours, varying (presumbly) with OpenAI's load factors. We include waiting periods between attempts, following OpenAI's recommendations. To obtain well-defined answers from the model, we found it necessary to set the temperature to 0, for all our experiments. The model is designed to allow a window of approximately 4K tokens; this limits the number of few-shot samples. For our experiments, we used 3 shots. $\mathcal{A}SAP$ defaults to three shots because related work [3, 12] has shown, and our own experiments with $\mathcal{A}SAP$ confirmed, that more shots did not significantly improve performance. However, for up to 2% of the randomly chosen samples in each experiment, we didn't get good results; either the prompt didn't fit into the model's window, or the model mysteriously generated an empty string. In cases where the prompt as constructed with 3 samples was too long, we automatically reduce the number of shots. When empty summaries were emitted, we resolved this by increasing the number of shots. This simple, repeatable, modest-overhead procedure can be incorporated into automated summarization tools.

[5]Please see Section 7 for the rationale.

4 RESULTS

We evaluate the benefits of $\mathcal{A}SAP$-enhanced prompts, for code summarization, in different settings and using various metrics. We find evidence of overall performance gain, in studies on six languages. However, for other detailed analyses, we focused primarily on Java and Python, because of OpenAI API rate limits.

4.1 Encoder-decoders & Few-shot Learning

Our baseline results on CodeSearchNet [47], using IR-based few-shotting, come first. Prior work reports that IR methods can find better samples for few-shot prompting, for tasks such as program repair [48] and code generation [34]. In Table 2, we observe that this is also true for code summarization; we note improvements of 3.00 (15.10%) and 1.12 (5.42%) in BLEU-4 score for Java and Python, respectively, simply by using *BM25* as a few-shot sample selection mechanism. Since *BM25* was already used in prior paper (albeit for other tasks) [48], we consider this *BM25*-based few-shot learning for code summarization as just a baseline (not a contribution *per se*) of this paper.

4.2 $\mathcal{A}SAP$ Prompt Enhancement

We now focus on the central result of our paper: the effect of $\mathcal{A}SAP$ prompt enhancement. Table 3 shows the component-wise and overall improvements achieved after combining all the prompting components for all six programming languages. BLEU improvements range from 1.84 (8.12%) to 4.58 (16.27%). In most cases, we see improvements of over 2.0 BLEU, the required threshold for human perception noted by Roy *et al.* [57].

We also noticed that all three components (i.e., *Repo*sitory Information., *DFG* Data Flow Graph, *Id*entifiers) help the model achieve better performance in all six languages, as we combined these components individually with *BM25*. However, for Ruby, the best performing combination includes just the Repo. information. In most cases, the Repo. helps a lot, relative to other components.

To ascertain improvement significance, we used the pairwise one-sided Wilcoxon signed-rank test, finding statistical significance in all cases for our final prompt when compared with vanilla BM25 few-shot learning, even after adjusting for false discovery risk.

4.3 Same Project Code Summarization

We now examine the benefits of $\mathcal{A}SAP$ in the context of some earlier work on few-shot selection. Prior work has shown that selecting few-shots from the same projects substantially improves performance [3]. To see if our prompt enhancement idea further helps in project-specific code summarization, we evaluated our approach on the dataset from Ahmed and Devanbu [3]. Due to rate limits, we reduced the number of test samples to 100 for each of the four Java and Python projects. Since we have too few samples for a per-project test, we combined all the samples to perform the

Language	CodeBERT	GraphCodeBERT	Polyglot CodeBERT	Polyglot GraphcodeBERT	CodeT5	CodeT5+	Few-shot (random)	Few-shot with BM25	Gain (%) over random few-shot
Java	18.8	18.52	20.22	19.94	19.78	19.83	19.87	**22.87**	+15.10%
Python	17.73	17.35	18.19	18.33	19.98	18.85	20.66	**21.78**	+5.42%

Table 2: Performance of encoder-decoder and few-shot models on Java and Python code summarization, measured using BLEU.

Language	BM25	BM25+repo	BM25+id	BM25+DFG	*ASAP*	Comparing with BM25	
						Gain (%) over BM25	p-value
Java	22.87	25.23	23.39	23.13	**25.41**	+11.11%	<0.01
Python	21.78	24.22	22.54	21.82	**24.26**	+11.39%	<0.01
Ruby	17.21	**19.67**	19.19	17.55	19.62	+14.00%	<0.01
JavaScript	23.27	25.11	24.21	24.04	**25.36**	+8.98%	<0.01
Go	22.67	24.41	23.2	23.42	**24.51**	+8.12%	<0.01
PHP	28.15	32.07	29.8	28.92	**32.73**	+16.27%	<0.01
Overall	22.66	25.12	23.72	23.15	**25.32**	+11.74%	<0.01

Table 3: Performance of prompt enhanced comment generation with code-davinci-002 model, measured using BLEU. p-values are calculated applying one-sided pair-wise Wilcoxon signed-rank test and B-H corrected.

Language	Project Name	#of training sample	#of test sample	Cross-project			Same-project		
				BM25	*ASAP*	p-value	BM25	*ASAP*	p-value
Java	wildfly/wildfly	14	100	24.05	**24.77**	<0.01	17.86	18.27	<0.01
	orientechnologies/orientdb	10	100	25.54	**27.23**		19.43	20.24	
	ngageoint/geopackage-android	11	100	29.33	42.84		45.48	**46.21**	
	RestComm/jain-slee	12	100	17.04	19.06		17.99	**19.61**	
Python	apache/airflow	12	100	20.39	20.37		20.36	**20.72**	
	tensorflow/probability	18	100	**21.36**	21.18		20.30	20.86	
	h2oai/h2o-3	14	100	19.50	**20.72**		18.75	19.81	
	chaoss/grimoirelab-perceval	14	100	25.23	29.23		32.75	**38.23**	

Table 4: Performance of prompt enhanced comment generation with code-davinci-002 model on same project data (measured using BLEU) and p-values are calculated applying one-sided pair-wise Wilcoxon signed-rank test after combining the data from all projects.

Language	# of Samples	Exact Match (EM)				Edit Similarity (ES)			
		Zero-shot	*ASAP* (Zero-shot)	Gain (%)	p-value	Zero-shot	*ASAP* (Zero-shot)	Gain (%)	p-value
Java	9292	20.75	**22.12**	+6.6%	<0.01	55.35	**59.66**	+7.79%	<0.01
Python	6550	14.05	**14.58**	+3.77%	0.13	49.71	**50.12**	+0.82%	<0.01
Overall	15842	17.97	**19.01**	+5.79%	<0.01	53.01	**55.72**	+5.11%	<0.01

Table 5: Performance of *ASAP* enhanced prompts with code-davinci-002 model on line completion task.

statistical test. Note that our total sample size for the statistical test exceeds the number of required samples determined through the analysis mentioned in Section 7. When working with the same project, one must split data with care, to avoid leakage from future samples (where desired outputs may already exist) to past ones. Therefore, we sorted the samples by creation dates in this dataset. After generating the dataset, we applied our approach to evaulate the performance in same project setting. We also compared our results with a cross-project setup, where we retrieved samples from the complete cross-project training set, similar to the setting used in Section 4.2.

Table 4 shows the results project-based code summarization. Note that this is a project-specific scenario where data is not available at all. The training data for each project is very limited. We found that, for 4 projects, cross-project few-shot learning yielded the best performance; while, for 4 others, same-project few-shot learning was most effective. We note that Ahmed & Devanbu didn't use IR to select few-shot samples and consistently achieved better results with same-project few-shot learning [3]. IR does find relevant examples in the large samples available for Java & Python, and we get good results. We analyzed 16 pairs of average BLEU from 8 projects, considering both cross-project and same-project scenarios. Our prompt-enhanced few-shot learning outperformed vanilla BM25 retrieved few-shot learning in 14 cases (87.5%). This suggests that *ASAP* prompt enhancement is helpful across projects. *ASAP* statistically improves performance in both cross-project and same-project settings.

4.4 Is *ASAP* Model-agnostic?

Our results so far pertain to the code-davinci-002 models. We also fed *ASAP*-augmented prompts to the other two models, text-davinci-003 & gpt-3.5-turbo (chat model). Our findings are in Table 6. Our prompt-enhanced few-shot learning approach improved the

Language	Model	BM25	*ASAP*	Gain	p-value
Java	Code-davinci-002	23.90	**25.78**	+7.87%	<0.01
	Text-davinci-003	18.98	**22.31**	+17.54%	<0.01
	Turbo-GPT-3.5	16.68	**16.96**	+1.68%	0.95
Python	Code-davinci-002	22.00	**24.78**	+12.64%	<0.01
	Text-davinci-003	16.74	**18.93**	+13.08%	<0.01
	Turbo-GPT-3.5	15.01	**16.38**	+9.13%	<0.01
PHP	Code-davinci-002	28.42	**33.52**	+17.95%	<0.01
	Text-davinci-003	21.67	**25.72**	+18.69%	<0.01
	Turbo-GPT-3.5	18.48	**19.99**	+8.17%	<0.01

Table 6: Performance on code summarization, measured using BLEU. p-values are calculated applying one-sided pairwise Wilcoxon signed-rank test and B-H corrected.

performance of the gpt-3.5-turbo model by 1.68% to 9.13% and text-davinci-003 model by 13.08% to 18.69% on 500 samples each from Java, Python, PHP.

Gpt-3.5-turbo does worse than the code-davinci-002 and text-davinci-003 models at code summarization. The Turbo version is verbose and produces comments stylistically different from those written by developers, and also from the few-shot exemplars in the prompt. Careful prompt-engineering might improve the turbo model and enable it to generate more natural, brief comments; this is left for future work. This underperformance by the chat model is consistent with the findings by Kocon et al. [39]. Text-davinci-003 model showed the maximum performance increase (albeit still outdone by code-davinci-002). Note that text-davinci-003 is a completion model, like code-davinci-002. Our findings suggest that *ASAP* is more effective with completion models than chat models. We also conducted pairwise one-sided Wilcoxon signed rank tests, and the statistical significance of our findings (except java with gpt-3.5-turbo) suggests that *ASAP* will apply beyond just the original code-davinci-002 model.

4.5 *ASAP* for Completion

Our primary focus so far has been on code-summarization, in a few-shot setting. Here, we explore if *ASAP* works on another task: *code completion*, in a *zero-shot* setting where no example is shown or presented to the model. We assessed the value of including semantic facts for the *line completion* task, where the model generates the next line given the prior line. We uniformly and randomly collected 9292 Java and 6550 Python samples from the CodeSearchNet dataset to conduct our evaluation. We randomly selected a line for each sample and tasked the model with generating that line, given *just all the preceding lines*. While applying *ASAP*, we append the repository information and other semantic facts (*i.e.*, tagged identifiers, DFG) *before* the preceding lines. Importantly, when generating tagged identifiers and DFG, we only used partial information from preceding lines to avoid information leakage from later lines to the target lines.

We used two metrics, Exact Match (EM) and Edit Similarity (ES), in line with the CodeXGLUE benchmark, to measure the model's performance. We conducted a McNemar test for EM and a pairwise Wilcoxon sign-rank test to evaluate the model's performance, similar to what we performed for code summarization. Table 5 summarizes our findings. We observe an overall 5.79% gain in Exact Match (EM) and a 5.11% gain in Edit Similarity (ES), highlighting the effectiveness of incorporating semantic facts. For Python, we find statistical significance only for ES improvement, not for EM.

Language	Prompt Component	BLEU-4
Java	ALL	**25.41**
	-Repo.	23.50
	-Id	25.27
	-DFG	24.86
Python	ALL	**24.26**
	-Repo.	22.80
	-Id	23.93
	-DFG	23.31

Table 7: Ablation study.

4.6 Performance on Other Metrics

In addition to BLEU-CN, we measured performance with 3 other metrics; BLEU-DC, ROUGE-L and METEOR. Our results, in Table 10, shows average gains with *ASAP* on all three metrics. We conducted pairwise one-sided Wilcoxon signed-rank tests and found significant performance improvements with BLEU-DC and ROUGE-L for all the languages. However, we did not observe significant differences with METEOR for 4 out of 6 languages, though sample averages do improve with *ASAP* in all 6 comparisons. It's worth noting that we had only 1000 language samples (due to cost) for each language, so it's not unexpected to see some cases where we didn't observe significance. To evaluate the overall impact of *ASAP*, we combined the dataset from all languages for code-davinci-002 model (6000 samples) and performed the same test; we then get statistical significance (p-value < 0.01) for all three metrics, suggesting that *ASAP* does provide value.

5 DISCUSSION AND ABLATION STUDY

We now present an ablation study of *ASAP*'s design and the particular semantic facts our instantiation of *ASAP* uses before comparing *ASAP*'s output to our vanilla *BM25* baseline. The primary aim of an ablation study is to gauge the contribute of each aspect of a model to the final observed performance In our study, we removed each semantic component of the enhanced prompt and observed performance. We found that the Repo. component contributes most to the model's performance (Table 7) both for Java and Python. However, tagged identifier and DFG are also helpful, and the best results were obtained when we combined all three components in the prompt.

Two Illustrative Examples When manually examining results, we observed that in several samples, the *ASAP* prompt contained information that was crucial for the summary. Table 8 shows two example results that illustrate this point. In the first example, the baseline model failed to generate the term "element-wise". However, our prompted enhanced version capture this important concept, yielding a higher BLEU-4 score of 74.0 compared to the baseline score of 39.0. Similarly, in the second example, the baseline model did not recognize the function as a standalone process, leading to a low BLEU score of 10.0. However, our proposed approach did identify the function as a standalone process, resulting in a higher BLEU score of 33.0.

Example 1

```
def round(input_a, name: nil)
    check_allowed_types(input_a, TensorStream::Ops::FLOATING_POINT_TYPES)
    _op(:round, input_a, name: name)
end
```

Gold & model output	Comment	BLEU
Gold	Rounds the values of a tensor to the nearest integer element - wise	NA
BM25	Round a tensor to the nearest integer	39
ASAP	Rounds the values of a tensor to the nearest integer, element-wise.	74

Example 2

```
public static void main(final String[] args) {
    loadPropertiesFiles(args);
    final ShutdownSignalBarrier barrier = new ShutdownSignalBarrier();
    final MediaDriver.Context ctx = new MediaDriver.Context();
    ctx.terminationHook(barrier::signal);
    try (MediaDriver ignore = MediaDriver.launch(ctx))
    {
        barrier.await();
        System.out.println("Shutdown Driver...");
    }
}
```

Gold & model output	Comment	BLEU
Gold	Start Media Driver as a stand - alone process .	NA
BM25	Main method that starts the CLR Bridge from Java .	10
ASAP	Main method for running Media Driver as a standalone process.	33

Table 8: Selected examples, illustrating the effectiveness of *ASAP* enhancement.

Does the Model Memorize the Path? Of the three semantic facts *ASAP* adds to a prompt, repo. information impacts the model's performance most. This may be due to the fact that Code-Davinci-002 had memorized the specific file paths in our data during pre-training; when we provide the path to the function, perhaps the model just recalls memorized information? To investigate this question, we change the path representation: we took the repository name and path, split the tokens at "/", and presented the model with a list of tokens. The main idea behind this approach is to diffuse the original representation, and present the model with something not encountered during pre-training. If the model isn't literally memorizing, its performance should not be impacted. We observed that the differences between both versions were very small. For Java, we gained 0.24 BLEU but, for Python, we lost 0.04 with tokenized paths. This suggests a lower risk that the model memorized the path to the function.

Is the Identifier Tag Necessary? In this paper, we assign roles to the identifiers and tag them as *Function Name, Parameters, Identifier etc.* in the prompt (See Figure 2). Does this explicit tagging actually help performance? To investigate this question, we compare the model's performance when provided with a plain, "tag-free" list of identifiers. We observed that the tagged identifiers lead to better performance for both Java and Python than a simple tag-free list of identifiers. Our performance metric BLEU increased by 0.41 and 1.22 for Java and Python, respectively, suggesting that explicit semantic information does indeed contribute to better model performance.

Language	Prompt Enhanced		Vanilla BM25	
	#of shots	BLEU-4	#of shots	BLEU-4
Java	3	**25.41**	3	22.87
			4	23.13
			5	23.20
Python	3	**24.26**	3	21.78
			4	21.89
			5	21.74

Table 9: Comparing with higher-shots Vanilla BM25.

What's Better: More Shots or ASAP? Despite having billions of parameters, LLMs have limited prompt sizes. For example, code-davinci-002 and gpt-3.5-turbo support allow prompt-lengths of just 4k tokens. *ASAP* augmentation does consume some of the available prompt length budget! Thus we have two design options: 1) use fewer, *ASAP*-Augmented samples in the prompt or 2) use more few-shot samples *sans* augmentation. To investigate this, we also tried using 4 and 5 shots (instead of 3) for Java and Python with the code-davinci-002 model. However, Table 9 shows that higher shots using BM25 does not necessarily lead to better performance. With higher shots, there is a chance of introducing unrelated samples, which can hurt the model instead of helping it.

Only for Java did we observe better performance with both 4 and 5 shots compared to our baseline model. However, our proposed technique with just 3-shots still outperforms using BM25 with 5 shots. It's worth noting that the context window of the model is increasing day by day, and the upcoming GPT-4 model will allow us to have up to 32K tokens[6]. Therefore, the length limit might not be an issue in the near future. However, our study suggests that Automated Semantic Augmentation will still be a beneficial way to use available prompt length budget; moreover, it stands to reason that constructing more signal-rich, informative prompts will beneficial regardless of length.

What's New in *ASAP*'s Output? We add a *pro forma* analysis of a few hand-picked examples, to be consistent with peer-review-required community rituals; however, these analyses are highly anecdotal must be interpreted cautiously. We manually examine several samples to discuss our results in greater detail; specifically, to answer three questions: to specify 1) the new types of information *ASAP* presents to the LLM and 2) how *ASAP*'s summaries differ from those created by existing techniques, and 3) to analyze the errors that *ASAP* introduces. Table 11 presents some samples where, for the first three, *ASAP* performed very well compared to our retrieval-based baselines, and for the second three, the baseline performed better than *ASAP*. While we discuss our findings in the context of the provided samples, our observations generalise to other samples.

The new types of information ASAP presents to the LLM: As discussed in the paper, our primary contribution involves *augmenting* retrieved samples (retrieved using BM25, as per Nashid et al. [48]) with *semantic facts*, resulting in improved performance compared to the base retrieval approach. We add semantic facts related to repository details, identifiers, and data flow graphs to both retrieved samples and input code. As anticipated, the added semantic facts transfer into, and enhance, the model output.

In the first sample, the baseline retrieval-only method fails to capture the term "gradient" entirely. However, by incorporating

[6] https://platform.openai.com/docs/models/gpt-4

Language	BLEU-DC				ROUGE-L				METEOR			
	BM25	ASAP	Gain (%)	p-value	BM25	ASAP	Gain (%)	p-value	BM25	ASAP	Gain (%)	p-value
Java	14.09	**15.94**	+13.13%	<0.01	36.85	**38.41**	+4.23%	<0.01	35.66	**36.10**	+1.23%	0.32
Python	12.63	**14.49**	+14.73%	<0.01	35.32	**37.74**	+6.85%	<0.01	33.05	**35.63**	+7.81%	<0.01
Ruby	9.16	**11.01**	+20.2%	<0.01	28.19	**30.55**	+8.37%	<0.01	27.65	**29.20**	+5.61%	0.03
JavaScript	14.89	**16.71**	+12.22%	<0.01	32.28	**33.88**	+4.96%	<0.01	32.08	**33.02**	+2.93%	0.15
Go	17.10	**18.57**	+8.60%	<0.01	41.04	**42.43**	+3.39%	<0.01	36.78	**37.26**	+1.31%	0.27
PHP	16.97	**20.63**	+21.57%	<0.01	40.48	**44.90**	+10.92%	<0.01	40.14	**43.35**	+8.00%	<0.01
Overall	14.14	**16.23**	+14.78%	<0.01	35.69	**37.99**	+6.44%	<0.01	34.23	**35.76**	+4.47%	<0.01

Table 10: The effectiveness of ASAP in popular code summarization metrics. p-values are calculated applying one-sided pair-wise Wilcoxon signed-rank test and B-H corrected.

Change (BLEU-4)	Reference	BM25	*ASAP*
ASAP (+47)	Generates a gradient - noise value from the coordinates of a three - dimensional input value and the integer coordinates of a nearby three - dimensional value .	Computes a value at the point (x, y, z) in a 3D Perlin noise function.	Generates a gradient-coherent-noise value from the coordinates of a three-dimensional input value.
ASAP (+26)	Replaces type with mapped type for current path .	Returns the mapped type for the current path, or the target type if no mapping exists .	Replaces target type with mapped type if any .
ASAP (+41)	convert a dataRootExt to a dataRoot	Convert to a DataRoot object .	Convert a DataRootExt to a DataRoot .
ASAP (-36)	Computes the convex hull of the set of points .	Computes the convex hull of the set of points	Computes the convex hull of the set of points using Andrew's monotone chain algorithm
ASAP (-35)	Getter for the Uninstall Actions .	Returns the uninstall actions .	Retrieves the uninstall actions for this deployable unit .
ASAP (-67)	Get a column of this matrix .	Get a column of this matrix .	Return the specified column of this matrix as a column vector .

Table 11: Examples Showing Strength and Weakness of *ASAP*.

semantic facts, the model successfully recovers the term because it is frequently found in both identifiers and repository names, influencing the model's output. In the second example, where the goal is to replace rather than simply return, the baseline fails to generate the term "replace", despite the clear indication in the function name ("replaceWithMappedTypeForPath"). The data flow between identifiers, provided in the semantic facts, may have helped the model recognize replacement operations.

How ASAP's summaries differ from those created by existing techniques: Following the above discussion, we observed that *ASAP* is generating more specific information:

(1) It identifies "gradient" in sample 1.
(2) It suggests changing "return" to "replace" in another sample (sample 2).
(3) It recommends changing "dataroot" to "datarootext" in a different sample (sample 3).

These differences were observed across multiple samples when comparing our baseline to *ASAP*. The *ASAP* approach consistently produces more specific information compared to the baseline.

Analyze the errors that ASAP introduces: The examined examples suggest that *ASAP* can become too specific, and thus not match the developer-written summary. *ASAP* gets over-specific in the last three examples with "Andrew's monotone chain algorithm" and "deployable unit", "column vector". While these terms are not necessarily incorrect, BLEU-4 drops, because the developer-written summary was more generic.

We also observe quantitatively that *ASAP* induced positive changes in 44% of the samples. However, the performance also declined for 30% of the samples, and remained the same on the rest. Compared to our baseline (few-shot learning with BM25-retrieved samples), *ASAP* requires more tokens. The additional token cost, per query (both in terms of monetary cost and performance overhead) is quite modest. On the other hand, we observe a substantial 12% overall improvement with *ASAP* using the Codex model.

6 RELATED WORK

6.1 Code Summarization

Deep learning models have advanced the state-of-the-art in SE tasks such as code summarization. The LSTM model for code summarization was first introduced by Iyer *et al.* [33]. Pre-trained transformer-based [62] models such as CodeBERT [21], PLBART [2], and CodeT5 [64] have been extensively used on the CodeXGLUE [47] code summarization dataset, resulting in significant improvements. However, there is a caveat to using pre-trained language models: although these models perform well, extensive fine-tuning is required, which can be data-hungry & time-consuming. Additionally, separate models had to be trained for different languages, increasing training costs. To reduce the number of models required, multilingual fine-tuning has been suggested, to maintain or improve performance while reducing the number of models to one [4]. However, this approach did not reduce the training time or the need for labeled data.

LLMs, or large language models, are much larger than these pre-trained models, and are trained on much bigger datasets with a simple training objective — auto-regressive next-token prediction [12]. These models perform surprisingly well on tasks, even without fine-tuning. Just prompting the model with different questions, while providing a few problem-solution exemplars, is sufficient. Few-shot learning has already been applied to code summarization, and has been found to be beneficial [3].

6.2 Other Datasets

There are several datasets available for code summarization, in addition to CodeXGLUE [47]. TL-CodeSum [30] is a relatively smaller dataset, with around 87K samples, but it does include duplicates, which may result in high performance estimates that may not generalize. Funcom [41] is a dedicated dataset with 2.1 million Java functions, but contains duplicates. We chose CodeXGLUE (derived from CodeSearchNet) because it is a diverse, multilingual dataset that presents a challenge for models. Even well-trained models like CodeBERT struggle on this benchmark; its performance is particularly poor on languages with fewer training samples.

There has been a lot of work on code summarization, ranging from template matching to few-shot learning. These models use different representations and sources of information to perform well in code summarization. Comparing or discussing all of these models is beyond the scope of this work. We note, however, that our numbers represent a new high-point on the widely used CodeXGlue benchmark for code summarization and code-completion; we refer the reader to https://microsoft.github.io/CodeXGLUE/ for a quick look at the leader-board. Our samples are smaller (N=1000), but the estimates, and estimated improvements, are statistically robust (See the sample size discussion in Section 7).

6.3 LLMs in Software Engineering

Although LLMs are not yet so widely used for code summarization, they are extensively used for code generation [14, 49, 67] and program repair [5, 18, 35, 36]. Models like Codex aim to reduce the burden on developers by automatically generating code or completing lines. Several models such as Polycoder [67] and Codegen [49] perform reasonably well, and due to their few-shot learning or prompting, they can be applied to a wide set of problems. However, Code-davinci-002 model generally performs well than those models and allows us to fit our augmented prompts into a bigger window.

Jain et al. proposed supplementing LLM operation with subsequent processing steps based on program analysis and synthesis techniques to improve performance in program snippet generation [34]. Bareiß *et al.* showed the effectiveness of few-shot learning in code mutation, test oracle generation from natural language documentation, and test case generation tasks [10]. CODAMOSA [42], an LLM-based approach, conducts search-based software testing until its coverage improvements stall, then asks the LLM to provide example test cases for functions that are not covered. By using these examples, CODAMOSA helps redirect search-based software testing to more useful areas of the search space. Jiang et al. evaluated the effectiveness of LLMs for the program repair problem [35].

Retrieving and appending a set of training samples has been found to be beneficial for multiple semantic parsing tasks in NLP, even without using LLM [68]. One limitation of this approach is that performance can be constrained by the availability of similar examples. Nashid et al. used a similar approach and gained improved performance in code repair and assertion generation with the help of LLM [48]. However, none of the above works has attempted to automatically semantically augment the prompt. Note that it is still too early to comment on the full capabilities of these large language models. Our findings so far suggest that augmenting the exemplars in the prompt with semantic hints helps on the code summarization and code completion tasks; judging the value of $\mathcal{ASAP}$ in other tasks is left for future work.

7 THREATS & LIMITATIONS

A major concern when working with large language models is the potential for test data exposure during training. Sadly, one can't directly check this since the training dataset is not accessible. The model's lower performance with random few-shotting suggests that memorization may not be a major factor. As we incorporate relevant information, the model's performance improves with the amount and quality of information. Had the model already memorized the summaries, it could have scored much higher, even without the benefit of relevant exemplars and semantic augmentation.

Sample Size Analysis: We used the observed means and standard deviations to calculate (using G*power [19, 20]) the required sample sizes, using commonly used values: α of 0.01 (desired p-value) and a β of 0.20 (*viz*, a 20% chance of NOT discovering an effect, should one exist). For the tests that we used (Wilcoxon Signed-rank test), we found that the needed sample size was always below the sample size we used for our primary studies, *viz.,* 1000.

User Study: We did not conduct a user study for $\mathcal{ASAP}$. Thus, the enhancements in metrics presented here may not necessarily translate into improved developer performance. This aspect is left to future work.

Finally: fine-tuning large LMs to use derived semantic facts may improve on our augmented prompting approach, but would be costly. We will leave its consideration to future research.

8 CONCLUSION

In this paper, we explored the idea of *Automatic Semantic Augmentation of Prompts*, whereby we propose to enhance few-shot samples in LLM prompts, with tagged facts automatically derived by semantic analysis. This based on an intuition that human developers often scan the code to implicitly extract such facts in the process of code comprehension leading to writing a good summary. While it is conceivable that LLMs can implicitly infer such facts for themselves, we conjectured that adding these facts in a formatted style to the exemplars and the target, within the prompt, will help the LLM organize it's "chain of thought" as it seeks to construct a summary. We evaluated this idea a challenging, de-duplicated, well-curated CodeSearchNet dataset, on two tasks: code summarization and code completion. Our findings indicate that Automated Semantic Augmentation of Prompts is generally helpful. Our estimates suggest it helps surpass state-of-the-art.

Acknowledgements: We would like to acknowledge National Science Foundation under Grant NSF CCF (SHF-MEDIUM) No. 2107592 and the Intelligence Advanced Research Projects Agency (IARPA) under contract W911NF20C0038 for partial support of this work. Our conclusions do not necessarily reflect the position or the policy of our sponsors and no official endorsement should be inferred.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.

[2] 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*. 2655–2668.

[3] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[4] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*. 1443–1455.

[5] Toufique Ahmed and Premkumar Devanbu. 2023. Better patching using LLM prompting, via Self-Consistency. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1742–1746.

[6] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models. *ICSE* (2023).

[7] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[8] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[9] 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*. 65–72.

[10] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335* (2022).

[11] Lionel C Briand. 2003. Software documentation: how much is enough?. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* IEEE, 13–15.

[12] 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 (2020), 1877–1901.

[13] Boxing Chen and Colin Cherry. 2014. A systematic comparison of smoothing techniques for sentence-level BLEU. In *Proceedings of the ninth workshop on statistical machine translation*. 362–367.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[16] Samanta Dey, Venkatesh Vinayakarao, Monika Gupta, and Sampath Dechu. 2022. Evaluating commit message generation: to BLEU or not to BLEU?. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 31–35.

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

[18] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. ICSE.

[19] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang. 2009. Statistical power analyses using G* Power 3.1: Tests for correlation and regression analyses. *Behavior research methods* 41, 4 (2009), 1149–1160.

[20] Franz Faul, Edgar Erdfelder, Albert-Georg Lang, and Axel Buchner. 2007. G* Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior research methods* 39, 2 (2007), 175–191.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*. 1536–1547.

[22] Andrew Forward and Timothy C Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*. 26–33.

[23] Jianfeng Gao and Xiaodong He. 2013. Training MRF-based phrase translation models using gradient ascent. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 450–459.

[24] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment ?Translation?: Data, Metrics, Baselining & Evaluation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 746–757.

[25] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[26] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 223–226.

[27] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.

[28] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 36–47.

[29] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.

[30] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 2269–2275.

[31] Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards Reasoning in Large Language Models: A Survey. *arXiv preprint arXiv:2212.10403* (2022).

[32] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[33] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[34] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings, 44th ICSE*. 1219–1231.

[35] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *ICSE* (2023).

[36] Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2022. Repair is nearly generation: Multilingual program repair with llms. *arXiv preprint arXiv:2208.11640* (2022).

[37] Hassan Kane, Muhammed Yusuf Kocyigit, Ali Abdalla, Pelkins Ajanoh, and Mohamed Coulibali. 2020. NUBIA: NeUral Based Interchangeability Assessor for Text Generation. arXiv:2004.14667 [cs.CL]

[38] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *ICSE* (2023).

[39] Jan Kocoń, Igor Cichecki, Oliwier Kaszyca, Mateusz Kochanek, Dominika Szydło, Joanna Baran, Julita Bielaniewicz, Marcin Gruza, Arkadiusz Janz, Kamil Kanclerz, et al. 2023. ChatGPT: Jack of all trades, master of none. *Information Fusion* (2023), 101861.

[40] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916* (2022).

[41] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[42] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th International Conference on Software Engineering, ser. ICSE*.

[43] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[44] Chin-Yew Lin and Franz Josef Och. 2004. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. 501–507.

[45] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[46] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

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

[48] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *Proceedings, 45th ICSE*.

[49] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language

model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[50] 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*. 311–318.

[51] Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. 2022. Reasoning with Language Model Prompting: A Survey. *arXiv preprint arXiv:2212.09597* (2022).

[52] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[53] 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 (2019), 9.

[54] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Citeseer, 29–48.

[55] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[56] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*. 390–401.

[57] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1105–1116.

[58] Thibault Sellam, Dipanjan Das, and Ankur P Parikh. 2020. BLEURT: Learning robust metrics for text generation. *arXiv preprint arXiv:2004.04696* (2020).

[59] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2023. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*. 1597–1608.

[60] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2022. Repository-level prompt generation for large language models of code. *arXiv preprint arXiv:2206.12839* (2022).

[61] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[63] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[64] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[65] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems* 32 (2019).

[66] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903* (2022).

[67] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[68] Yury Zemlyanskiy, Michiel de Jong, Joshua Ainslie, Panupong Pasupat, Peter Shaw, Linlu Qiu, Sumit Sanghai, and Fei Sha. 2022. Generate-and-Retrieve: use your predictions to improve retrieval for semantic parsing. *arXiv preprint arXiv:2209.14899* (2022).

[69] 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*. 1385–1397.

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