DocGen: Generating Detailed Parameter Docstrings in Python

Vatsal Venkatkrishna*
vatsalvenkatkrishna@gmail.com
Australian National University
Australia

Durga Shree Nagabushanam*
durgashree.n15@gmail.com
Australian National University
Australia

Emmanuel Iko-Ojo Simon
ammanuel.Simon@anu.edu.au
Australian National University
Australia

Melina Vidoni
melina.vidoni@anu.edu.au
Australian National University
Australia

ABSTRACT

Documentation debt hinders the effective utilization of open-source software. Although code summarization tools have been helpful for developers, most would prefer a detailed account of each parameter in a function rather than a high-level summary. However, generating such a summary is too intricate for a single generative model to produce reliably due to the lack of high-quality training data. Thus, we propose a multi-step approach that combines multiple task-specific models, each adept at producing a specific section of a docstring. The combination of these models ensures the inclusion of each section in the final docstring. We compared the results from our approach with existing generative models using both automatic metrics and a human-centred evaluation with 17 participating developers, which proves the superiority of our approach over existing methods.

CCS CONCEPTS

• **Software and its engineering → Documentation**; • **Computing methodologies → Machine learning**; **Natural language generation**.

KEYWORDS

Docstrings, Pre-trained models, Code summarization, Scientific software

*Both authors contributed equally to this research.

1 INTRODUCTION

Documentation is a vital aspect of software development, but it is often neglected by developers because of its tedious and time-consuming nature [49]. As a result, documentation becomes incomplete, outdated or non-existent over time [54], leading to various problems such as increased maintenance costs, reduced software quality, and lower user satisfaction [1]. This is particularly concerning for scientists and mathematicians who develop software tailored to their specific goals. Since they are rarely trained in software engineering and coding best practices, they tend to leave poor documentation for "scientific software" [40, 48]. There is a tremendous increase in the inclusion of software applications for scientific research [30], simultaneously requiring extensive documentation to be useful and reproducible [37]. In the context of open-source software, method-level documentation (e.g., Python's docstrings) is crucial, as it enables the effective utilization of the software for further development and maintenance [42]. Python is one of the most widely used programming languages in computational science due to its versatility, simplicity and efficiency [3, 34].

Over the years, source code summarization [17, 24, 25] has been studied extensively, aiming to produce a high-level summary of a given code snippet. Several pre-trained language models specifically developed for code understanding tasks [13, 15, 57] have shown promising results on code summarization and a significant improvement over classical text-only summarization algorithms [32]. However, there have been few studies that analyse these algorithms' capability of generating a "detailed docstring" — one that describes the entire function holistically, with all the technical details of its parameters, which is essential for understanding and reusing code [51]. This is significantly more challenging than code summarization since it requires an in-depth understanding of each individual parameter's role in the function, as well as their interactions with other variables. Moreover, human-written documentation is often deficient [22], making high-quality training data scarce and limiting the performance of generative models. Therefore, complex documentation needs such as thoroughly explained parameters in dynamically typed languages remain an open challenge.

In this work, we present an approach to generate documentation catering to the specific needs of scientific

software. We based our approach on the paper by Vidoni and Codabux [55], which developed a taxonomy of parameter-level documentation directives[1]. Using this taxonomy as a baseline, we devised a composite framework with multiple modules, each addressing a different documentation directive. We compared our results to those obtained from using a single transformer-based model through automatic scoring methods like BLEU [36] and METEOR [4] and further validated our approach through a survey with 17 participating developers.

The rest of this paper is organized as follows. We present prior work that inspires and closely relates to our own (§2). We then formalize our research questions (§3.2) and describe the data-gathering steps along with an inclusion-exclusion criterion (§3.3). We then detail the issues observed in training data (§3.4) and subsequent preprocessing steps (§3.5). We then describe our approach of independently generating each directive and the design choices we made (§3.6). Finally, we discuss the results we obtained (§4) and its implications (§5), along with threats to validity (§6) and a conclusion (§7).

2 RELATED WORKS

Deep learning for source code summarization. Code summarization is the task of generating a natural language description, typically a single line, for a code snippet describing its purpose and functionality. Early approaches treated source code as plain text and employed machine translation algorithms to generate summaries [19, 58]. However, programming languages have complex syntax and rich structure, which make them fundamentally different from natural language. Several studies have employed methods to exploit the structural and semantic information of source code through Abstract Syntax Trees (ASTs) [2, 18, 21, 60], Control Flow Graphs (CFGs) [6] and Data Flow Graphs (DFGs) [16] for code summarization. In recent years, models pre-trained jointly on code and natural language have been very successful, with CodeT5 [57], ProphetNet-Code [43], CoTexT [39], SPT-Code [33] and UniXcoder [15] being some of the best performing models for code summarization [32].

Docstring generation. Cui et al. [11] introduced the task of "code explanation generation," which aims to generate an informative and detailed summary of the code. They provided an annotated dataset as well as a curriculum learning pipeline, achieving promising results. However, their codebase was not reproducible at the time of performing our experiments (May 2023), with the dataset they collected being inaccessible and irreproducible due to missing files and a lack of documentation. Moreover, their approach does not guarantee the generation of important docstring directives that we address through our approach. Clement et al. [10] developed an approach inspired by the T5 architecture [45] and achieved promising results on generating good quality docstrings, but similarly lack details on reproducibility.

Method-level documentation. Barone and Sennrich [5]

developed a large corpus of Python code-docstring pairs by mining open-source repositories on GitHub and trained on NMT models to obtain baseline results. Sulír and Porubän [52] approached method documentation by obtaining examples for parameters, returns and object states before and after method execution and converted the essence into natural language. Among the specific directives we cover in this paper, to the best of our knowledge, only datatype prediction has been extensively studied in prior work. Luo et al. [23] proposed a method, training a classifier for type prediction. Pradel et al. [41] proposed a similar neural classifier with an additional framework to ensure compatibility of predicted types. Peng et al. [38] integrated static prediction with deep learning through a dependency graph generated for an entire given repository. However, all these methods rely on both existing documentation and source code for producing predicting datatypes, which introduces a cyclic dependency on documentation. Mir et al. [28] developed a hierarchical neural network for type inference through similarity learning.

Taxonomies for documentation. The increasing number of taxonomies has contributed to the maturing of the field of Software Engineering [53]. Taxonomies on software documentation contain a detailed account of directives for developers to include in their repositories. Embedding such a taxonomy into docstring generation tools could improve the quality of documentation generated in terms of technical details and consistency. Monperrus et al. [29] performed an empirical study of projects in Java to develop a taxonomy of API directives. They introduce the "method call" sub-directive, which specifies the directives of a method to be included in API documentation. Vidoni and Codabux [55] developed a taxonomy for R documentation, which specifically classifies the directives for method-level documentation.

Novelty of our approach. We propose a multi-step fine-tuning approach to produce detailed docstrings documenting every parameter in a function while ensuring the inclusion of four directives – a description of the parameter's overall purpose, its datatype, its default value, and its acceptance of a "None" value. Since we used different models for each of these directives, we were able to use only those models that were best suited to each directive's need. Our choice of these directives is informed by a filtered version of the taxonomy introduced by Vidoni and Codabux [55], covering the most important and challenging directives presented. However, the aforementioned taxonomy was designed specifically for scientific software. Since different fields of software would benefit from a different set of directives being documented, we limited the scope of our study to scientific software to avoid presuming the nature of developers' needs at large. We conducted experiments to verify the novelty of our approach through automated scoring methods like BLEU [36] and METEOR [4] as well as through a survey of 17 experienced developers who compared our results with existing fine-tuning approaches.

---

[1]A documentation directive is a natural-language statement to inform developers of constraints and guidelines related to the correct usage of a section of code; in our case, a function's parameter

3 METHODOLOGY

3.1 Directives to Document

A taxonomy of documentation directives could be very helpful in guiding best practices. However, to our knowledge, there exists no taxonomy that describes the details to be documented for Python functions. However, such a taxonomy does exist for R [55]. Considering similar aspects of both Python and R, like dynamic typing and seamlessly blending object-oriented programming and functional programming, we decided to use this taxonomy as a guide for our models. To maintain simplicity in our work, we selected four directives from the taxonomy based on the availability of high-quality training data, namely:

- PD: a description of the general purpose and functionality of a parameter
- PV: the default value of a parameter
- PT: the default datatype of a parameter
- PN: Whether or not a parameter can take on a "None" value (the original taxonomy tracks both "Na" and "null" in R, which is equivalent to Python's "None").

Other directives, like exceptions that could be raised and the data formats of primitive and non-primitive datatypes, are equally crucial to a detailed docstring but were rarely included in our datasets. Hence, we chose to use the directives where data is either available or could be easily extracted.

3.2 Research Questions (RQs)

RQ1. Can each directive be produced using a separate model best suited to the task's complexities? Parameters are an important part of modern programming, especially in dynamically typed languages like Python, allowing users to reuse a function multiple times with varying inputs. Hence, it is crucial to document them systematically to allow for maintenance and reusability. We investigate the best approaches to learning the intricacies of generating the directives described in §3.1.

RQ2. Is a combination of task-specific models a more effective approach for docstring generation compared to a single model for all directives? For a single generative model to reliably produce all required docstring directives, the corresponding training data must contain a majority of instances with these directives present. Unfortunately, such code-docstring pairs are lacking in quantity and hard to enforce without manual annotation. Thus, we propose a multi-step fine-tuning method that guarantees the generation of these directives and compare it with existing single-model approaches.

RQ3. Do developers prefer the docstrings generated by our proposed multi-step approach over a single model's docstring? Developers are the primary benefactors of the software documentation. Thus, their opinions on the docstrings generated by our approach are highly valued. We conducted a survey with a Likert-based scoring strategy to assess their opinion on the docstrings from our multi-step approach and contrast them with the docstrings generated by a single generative model.

3.3 Data Gathering

Inclusion-Exclusion Criteria We defined inclusion and exclusion criteria (IEC) to determine suitable repositories for our study; they are summarised in Table 1.

Table 1: Inclusion and exclusion criteria (IEC) for selecting relevant software repositories.

Code	Inclusion Criteria
I1	Repositories with documentation written in English
I2	Repositories comprising at least 75% Python code
I3	Repositories with licenses among Apache license 2.0, Creative Commons license family, BSD Zero-Clause license, GNU General Public License family, MIT license
I4	Scientific software

Code	Exclusion Criteria
E1	Repositories with at least one commit after 1st Feb 2023
E2	Functions with no parameters
E3	Functions with no return variables
E4	Jupyter notebooks
E5	Forked repositories

We considered repositories with English as their primary language [I1] to avoid loss of semantics during translation. To limit the number of functions with dependencies on files written in other programming languages, we ensured that the repositories under consideration constituted at least 75% of all code written in Python [I2]. To avoid breaches in copyright regulation, we only mined repositories with selected licenses [I3]. We only included repositories that have one of (`data-science, machine-learning, deep-learning, statistics, and science`) listed as topics to restrict our study to scientific software [59] [I4].

We considered inactivity as an exclusion criterion [E1] to ensure the possibility of submitting a pull request to repositories lacking any documentation for extended validation on the docstrings generated if required. We also excluded the functions that do not accept parameters or return any value [E2, E3] due to their prominent role in a docstring [55]. We further excluded Jupyter notebooks due to differences in their documentation standards [14] [E4] and forked repositories to avoid duplication [E5]. We did not place any restrictions on the accompanying documentation in the IEC to allow for a testing set.

A dataset of scientific software. We used the dataset introduced by Biswas et al. [7], which contains 1558 repositories that use a diverse set of data science libraries. However, we still needed to check these repositories against our inclusion-exclusion criteria. We employed a two-stage approach for this check. First, we extracted the metadata pertaining to our IEC for all 1558 repositories using `libraries.io`[2] because of its high rate limit (60 requests per minute). However, the API was unable to retrieve the data for many existing repositories. Therefore, we used GitHub's

---

[2]https://libraries.io/api

REST API[3], which has a lower rate limit (1000 requests per hour), to retrieve the repositories that failed in `libraries.io`. After filtering, we were left with 873 repositories that use libraries related to data science and fit our IEC.

However, these 873 repositories had a very small number of repositories listing topics like `science` and `statistics`. Therefore, we manually mined repositories to represent other forms of scientific software. We applied our IEC through GitHub's Advanced Search Engine to search for repositories. This search retrieved 2422 repositories, but on closer inspection, we found a substantial number of false positives. Thus, two authors manually analyzed 200 of the retrieved repositories with the most stars and classified them on the basis of compliance with IEC. The authors individually provided a verdict of "one" indicating compliance and a "zero" otherwise for each repository. We achieved a Cohen's Kappa score of 0.779, denoting high inter-rater reliability between the authors to include 109 of the 200 repositories into the final dataset, resulting in a total of 982 repositories.

Extracting functions and docstrings. We extracted all the functions from our dataset of 982 repositories using `function-parser`[4], resulting in 111k functions with their corresponding docstrings. However, not all of these docstrings contained documentation corresponding to each parameter. Thus, we filtered these to include docstrings which contain at least one of the following tokens – `":param"`, `":arguments"`, `":args"`, `":parameters"`, `"param:"`, `"arguments:"`, `"args:"`, `"parameters:"`, since they are commonly observed in docstring conventions. Although this approach may have led to a few false negatives, it ensured the absence of false positives, thus maintaining the quality of our dataset. This filtering step leaves us with $41,462$ functions, which we hereby refer to as the *Raw* dataset.

3.4 Issues with Existing Docstrings

We randomly sampled 50 human-generated docstrings from our *Raw* dataset to identify the best data preprocessing strategies forward. We made the following observations on the issues with human-generated documentation, which supported the need for our multi-step approach to generate parameter docstrings:

- The amount of information documented for each parameter is not constant – In *Example Function 1* in Figure 1, *param2* has an "optional" attribute, indicating the presence of a default value, whereas the default value status of *param1* is not addressed.
- Some technical details are mentioned as part of the general description – In *Example Function 2* of Figure 1, the "None" status of *param1* and its corresponding action is mentioned as part of the short summary of the docstring and is left out when parameter descriptions alone are used for training.
- The position in which the datatype of a parameter is

Figure 1: Two sample docstrings, highlighting much of the issues we found in the *Raw* dataset. The docstrings differ in format and in the amount of information documented, a problem that becomes harder to deal with an increasing number of samples.

mentioned in the docstring may not be consistent – In Figure 1, the datatype of parameters is mentioned in parenthesis (*Example Function 1*), mentioned in the description or is not mentioned at all (*Example Function 2*).

A single generative model might struggle with these issues because it has to decide what details to include or exclude without any guidance. A multi-step approach solves this problem by using specific directives that ensure the inclusion of certain information. Producing a single directive is easier than producing a whole docstring because of persistent documentation debt. Moreover, each directive has its own intricacies, which a model trained for that specific directive could learn better.

3.5 Data Preprocessing

The following subsections detail our preprocessing approach and are summarized visually in Figure 2

Standardizing the data format. Docstrings can follow several writing conventions (e.g., ReST, Google, Numpydoc-style), and a mix of different styles in the data could negatively affect our models. Prior work suggests that the content documented takes precedence over the format it is presented in [46]. Therefore, to ensure formatting consistency in the training data, we cleaned up the *Raw* dataset by discarding newline tokens, extra spaces and external links/email addresses. We also removed examples enclosed by some docstrings along with long descriptions to ensure focus on parameters.

Further, we extracted the names, descriptions and datatypes of all parameters present in the docstring using `docstring-parser`[5]. This resulted in a dataset constituting

Legend

```
function code
docstring
```

```
def sample (p1, p2):
    < function body here >

--- short description ---

------------------------------
---- long description ----
------------------------------

:param p1: <desc1>
:param p2: <desc2>
:return r: <desc>
:raise <error>

Example:
------------------------------
------------------------------
------------------------------
Reference: abc.com.
Contact: abc@123.com
```
Raw Data

```
def sample (p1, p2):
    < function body here >

:param p1: <desc1>
:param p2: <desc2>
```
Formatted data

```
parameter 1:
def sample (p1, p2):
    < function body here >
:param p1: <desc1>
```

```
parameter 2:
def sample (p1, p2):
    < function body here >
:param p2: <desc2>
```
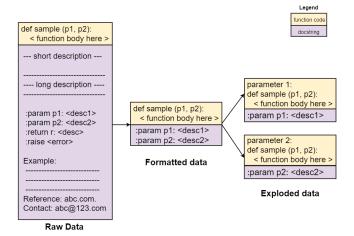**Exploded data**

Figure 2: An example of the preprocessing of *Raw* data to obtain the *Formatted* and *Exploded* datasets on a single instance. We discarded contacts, external links, short and long descriptions, return descriptions and potential errors raised from the *Raw* data and unified all parameter docstrings under a single format to obtain the *Formatted* data. We then split the instance into its constituent parameters and prepend the phrase "parameter N" before each code snippet.

parameter names and their corresponding docstrings.

Discarding partially documented functions. Upon closer inspection of our data, we found that some functions were only partially documented. For example, consider the following function and its docstring *def precook(s, n=4, out=False) ":param s: string, sentence to be converted into n-grams. :param n: int, number of n-grams for which representation is calculated."*. The description for the third parameter, "out", is not included. Therefore, we developed an algorithm to extract the parameters from function headers and compared it to the list of parameters extracted by `docstring-parser`. The results were ranked as follows:

- **Rank one:** The number of parameters extracted by `docstring-parser` is equal to the number of parameters extracted by our parameter extractor.
- **Rank two:** The number of parameters extracted by `docstring-parser` is greater than zero and lesser than the number of parameters extracted by our parameter extractor.
- **Rank three:** The number of parameters extracted by `docstring-parser` is zero, and the number of parameters extracted by our parameter extractor is greater than zero.

We only considered the functions ranked one for further pre-processing as these followed the docstring writing conventions [8], promising a better quality of data. We denote this dataset as the *Formatted* dataset, which we used as a benchmark for the best possible performance through a single model.

"Exploding" the data. Our proposed multi-step approach

generates the documentation for one parameter at a time, thereby requiring the training data for our models to be re-formatted accordingly. We split the *Formatted* dataset into N data points for each existing function, where N is the number of parameters in the corresponding function. To avoid different outputs for the same code snippet, we prepended the phrase "parameter N: " to the code string in each entry as shown in Figure 2 to indicate the parameter being documented in the docstring. This decomposition had a dual effect of simplifying the task as well as increasing the amount of training data. However, as highlighted in §3.4, these descriptions sometimes contained auxiliary information which, while seemingly harmless, could confuse the model into learning a pattern of these inclusions that doesn't actually exist. Thus, we restricted each entry in the dataset to only the first sentence. We hereby refer to this dataset as the *Exploded* dataset because it was obtained using the `pandas.DataFrame.explode` module[6]. Both the Formatted and Exploded datasets were split into training and validation sets, with 20% data in the training set. All fine-tuning is performed on the respective training sets, and results are on the respective validation sets.

3.6 Model Preparation

Pre-trained language models (PLMs) are deep learning models with millions, often billions, of parameters that can be fine-tuned to perform a variety of downstream tasks [56]. PLMs have also been applied to a number of code-understanding tasks to great success [31] like code summarization, which generates a single-line summary of a code snippet highlighting its purpose [13, 15, 57]. This enables us to fine-tune PLMs to perform our task of docstring generation for parameters. To our knowledge, very few studies have investigated the performance of these models in generating a detailed docstring, which is a much harder task than code summarization, and none have proposed a similar multi-step approach. We excluded Large Language Models (LLMs) from this analysis to limit the scope of our study to a multi-step approach and not digress into a debate about the benefits and demerits of using LLMs. We intend to perform an in-depth comparison of our approach with popular LLMs like GPT-4 [35] in a future study. Below, we describe our approach to producing each directive in detail.

Parameter Descriptions (PD). The task of generating parameter descriptions is similar to code summarization but requires an understanding of each parameter's role in the function. We experimented with three PLMs that are trained jointly on natural and programming languages:

- CodeBERT: Feng et al. [13] were the first to present a bimodal architecture, trained jointly on both natural and programming languages. It does not consist of a Transformer decoder, which must be trained from scratch for generative downstream tasks like code summarization. We included this model in our study

---

[6]https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.explode.html
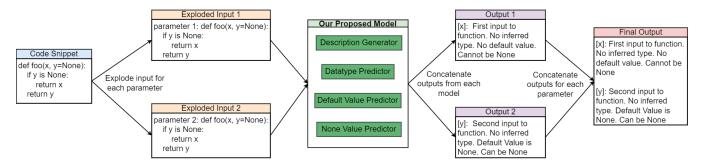
Figure 3: The pipeline for our proposed multi-step approach. Each input code snippet is replicated N times, where N is the number of parameters. Each replication is tagged with its corresponding parameter number. Each of these replications was then passed through the multi-model pipeline, comprising a description generator, datatype predictor, default value predictor, and None acceptance classifier. We concatenated the outputs from these models to produce our final output.

as a baseline to compare with other models.

- CodeT5: Wang et al. [57] extended the T5 model architecture [44] to code-understanding and generation tasks. We included this model in our study because of its proven excellence in code summarization [32]
- UniXcoder: Guo et al. [15] used mask attention matrices with prefix adapters to improve the representation of code using ASTs and code comments. We included this model in our study because it has achieved state-of-the-art in several code-related tasks [32].

We used the publicly available checkpoints on HuggingFace for all three models in our experiments, trained for 10 epochs with all hyperparameters at their default values. We then trained these models on the *Exploded* dataset with early stopping based on the BLEU score [36].

For a baseline comparison, we trained these models on the *Formatted* dataset, which is not instructed to generate any specific directives. This helped us assess the ability of current generative models to produce detailed docstrings and contrast their outputs with those from our multi-step finetuning approach.

Parameter Datatypes (PT). In the absence of good documentation, predicting each input parameter's datatype is a challenging task, requiring an in-depth understanding of their roles and interactions with other variables. Most prior work used existing documentation [23, 41] and other files in the same repository [38] to generate datatypes. Since we wanted to generate this documentation itself, we chose algorithms that use only the source code for predicting datatypes. To the best of our knowledge, there exists only one such algorithm – Type4Py [28]. It provided an API for type prediction, whose response contained data types of all parameters and variables used in the function. We extracted the predicted data types and mapped them with the corresponding parameters.

Parameter Default Values (PV). Identifying default values is a relatively simple task, especially in Python, where defaults are mostly declared in the function header itself. Thus, we implemented a simple AST-based algorithm to parse the

source code and infer defaults. We used Python's inbuilt `ast` module[7] to parse the input function and return an `ast.arg` object. If there are any default values for the parameters listed in `args`, the `defaults` attribute holds them. For keyword-only arguments (listed in the `kwonlyargs` attribute), the corresponding defaults are listed in the `kw_defaults` attribute. We extracted these default values.

Parameter None Acceptance (PN). We used a binary classification objective to classify a parameter's acceptance of the "None" value. We modified our *Exploded* dataset to include the classification label. If the full description (beyond one line) of the parameter contained the "None" token, a label of 1 was assigned, and 0 otherwise. This resulted in an unusually large number of negative instances, which we balanced out by sampling an equal number of positive and negative instances. We fine-tuned UniXcoder and CodeBERT on the resulting dataset for 10 epochs with a learning rate of 1e-5. Although CodeT5 is capable of performing classification-based tasks [57], we excluded it from this part of our study to maintain a comparison between encoder-only models. We evaluated the model with our validation set using the micro F1 score.

Final Output. The output from each of the modules described above was concatenated to obtain the docstring of a single parameter. The docstrings of each parameter were further concatenated to obtain the final output of our multi-step approach.

4 RESULTS & DISCUSSION

This section describes the results obtained from our experiments and also discusses lessons learned. We present our analysis for each RQ as follows:

4.1 RQ1. Multiple models tailored to generate specific directives can produce a detailed docstring.

The training data is a collection of human-written documentation which, as expected, is prone to numerous

---

[7]https://docs.python.org/3/library/ast.html

inconsistencies, as discussed in §3.4. Therefore, we hypothesized that a single model couldn't produce docstrings with certain directives reliably generated. Our proposed approach leveraged the strengths of multiple models of varying complexity that generated their respective directives and, hence, a detailed docstring. This section focuses on evaluating the outputs from each module of our approach from fine-tuning on the *Exploded* dataset with a multi-step approach.

Parameter Descriptions (PD). For evaluating parameter descriptions, we used BLEU [36] and METEOR [4], which are commonly employed to evaluate summarization-oriented tasks. They use different measures to compute the overlap between the generated and target texts. To apply these metrics, we used HuggingFace Evaluate[8]. These scores are reported in Table 2 under the "Exploded" subscript.

Table 2: Results for generating parameter descriptions. Subscripts indicate the dataset each model was trained on. The best result in each category is highlighted in bold, and the overall best model is underlined.

Model Name	BLEU-4	METEOR
CodeBERT $_{\text{Formatted}}$	0.212	0.476
CodeT5 $_{\text{Formatted}}$	**0.281**	**0.482**
UniXcoder $_{\text{Formatted}}$	0.224	0.462
CodeBERT $_{\text{Exploded}}$	<u>0.310</u>	<u>0.506</u>
CodeT5 $_{\text{Exploded}}$	0.278	0.485
UniXcoder $_{\text{Exploded}}$	0.254	0.465

We observed that CodeBERT significantly outperforms the remaining models, followed by CodeT5. This is a surprising result since CodeT5 has a decoder that is pre-trained on code-understanding tasks, while CodeBERT's decoder is fine-tuned from scratch, suggesting CodeT5's pre-training does not help in generating parameter-wise descriptions as it does in code summarization. Figure 4 shows a sample of the outputs generated by these models is in. The outputs of CodeBERT and CodeT5 were comparable, and the informal tone of CodeBERT's output for the `content` parameter was noteworthy. UniXcoder generates relatively bland outputs. We also observed that it points to wrong choices for the `mode` parameter in the description.

Parameter None Acceptance (PN). We used the micro F1-score, a commonly used metric for binary classification [50]. UniXcoder achieves a score of 0.704, followed by CodeBERT with 0.669. This behaviour is expected, since UniXcoder has outperformed CodeBERT on There is further scope for improvement, which could be achieved using a manually curated dataset instead of the synthetic one used in our approach. This will be discussed in Section 4.1.

Parameter Default Values (PV) and Datatypes (PT). We could not use any automatic metrics for evaluation to evaluate both parameter default values and datatypes due to the

---

```
Code Snippet

def write_content(filename, mode=None, content='happy'):
    if mode is not None:
        f = open(filename, mode)
    else:
        f = open(filename, 'w')
    f.write(content)
```

Output generated by CodeBERT

:param filename: name of file.
:param mode: mode in which to write the file.
:param content: Whatever you want to say...

Output generated by CodeT5

:param filename: path to the file.
:param mode: The mode to write the content to.
:param content: The content to write.

Output generated by UniXcoder

:param filename: path to a file.
:param mode: one of 'rb' or 'wb'.
:param content: string.

Figure 4: An example of the parameter descriptions generated from our multi-step approach using three different models.

absence of a labelled ground truth. Thus, we resorted to a human-centred evaluation strategy. We applied a sample size calculation of 95% confidence and 5% error to determine a representative set for evaluation, sampling 369 out of 9112 instances in our validation set. A subset of 369 was then randomly sampled and inspected simultaneously by two authors independently. Both authors inferred the datatypes and default values of the parameters indicated in the source code by themselves. If their verdict matched the model's, a positive score was given. We used Cohen's Kappa coefficient to measure the level of agreement between the reviewers. Although the approach was neither comprehensive nor extensive, it has a negligible risk of researcher bias. In future works, more in-depth studies must be conducted on generating and evaluating default values and data types.

The default values achieved a high accuracy of 93.7%, with a Cohen's Kappa of 0.903. To understand why the accuracy falls short of 100%, consider the following case: *def sample3(x, y=None): if y is None: y=5*. In this case, the default value of y is actually 5 and not "None". The presence of a few such samples, which are not standard Python coding practices, led to the error.

The datatype prediction achieved a significantly worse result, with an accuracy of 20% and a Cohen's Kappa of 0.828. This was expected because predicting default values is a significantly simpler task than predicting the data types of a variable, especially in Python, which is dynamically typed. These results indicate the need for more annotated datasets for datatype prediction.

Lessons Learned The performance of PLMs for specific natural language processing tasks depends on the quality of data they are fine-tuned on [27]. Documentation debt in scientific software repositories hinders us from developing all-rounder models which can include technical details along

with descriptions. To establish an approach that can tackle the existing debt and still produce the required output, we split the documentation task into smaller exclusive tasks. We observed that producing individual directives was not necessarily a generative task and varied significantly in complexity. PNs were modelled as a binary classification task with a synthetic dataset, PVs performed best with a simple rule-based method, and PTs used Type4Py, which is a hierarchical neural network architecture that predicts type clusters to produce a wide variety of datatypes.

RQ 1

> Models fine-tuned on code structures to produce descriptions for parameter docstrings were not consistent in the number of technical details included. Therefore, we combined the outputs from task-specific models to ensure the inclusion of the different technical directives of a parameter along with the description.

Future Works The approach to evaluating parameter datatypes was neither comprehensive nor extensive, as is expected in the absence of a well-labelled dataset. The performance of multiple models in the multi-step approach was evaluated individually. A metric to assess the approach as a whole could be of interest for further improvement.

4.2 RQ2. A combination of task-specific models is more effective than using a single model.

We compared our approach of using multiple models to a single-model approach for parameter docstring generation. The outputs generated by a single generative model fine-tuned on the *Formatted* dataset were compared to those obtained from our multi-step approach. For the single-model approach, we fine-tuned CodeBERT, CodeT5 and UniXcoder to generate the entire ground-truth docstring, as opposed to the parameter-wise input in our multi-step approach. The BLEU and METEOR scores of these approaches are shown in Table 2. We saw a significant improvement in CodeBERT's performance, almost 46% in BLEU-4 scores. UniXcoder also saw an improvement, although not as impressive, and CodeT5 had a very minor change in performance, with a minor decrement in its BLEU-4 score. However, these results are not sufficient to suggest the superiority of our approach since it does not account for the presence of the other modules used in our approach (i.e., PV, PN and PT). Moreover, the datasets used for fine-tuning these models are also different, with the *Exploded* dataset having fewer tokens than the *Formatted* dataset, and hence a lower scope for error.

Figure 5 showcases the best output from both approaches on the same code input. We observed that the best multi-step model tended to be more descriptive and informative than a single model. However, the outputs for PT and PN are not ideal because *filename* and *content* cannot be "None" without throwing an exception, and the type for *content* should also be *"Optional[str]"*. Nonetheless, we argue that the multi-step output is better overall since the single-model approach only

mentions the strings "None" and "Default" 3.2% and 1.9% of the time, respectively. This makes the multi-step approach more capable and reliable in generating detailed docstrings, with a future scope for improvement in its individual parts.

Code Snippet

```
def write_content(filename, mode=None, content='happy'):
    if mode is not None:
        f = open(filename, mode)
    else:
        f = open(filename, 'w')
    f.write(content)
```

Best output from formatted training (CodeT5)

:param filename: path to the file to write to.
:param mode: file mode to write to.
:param content: content to write.

Best output from exploded training (CodeBERT)

:param filename: name of file Type: str. No default value. Can be None.
:param mode: mode in which to write the file Type: Optional[str]. Default value is None. Can be None.
:param content: Whatever you want to say... Type: str. Default value is 'happy'. Can be None.

Figure 5: Comparison of outputs generated by the two approaches on the same example. The best model in both cases is showcased.

Lessons Learned The multi-step fine-tuning allows the model to relate each parameter individually with its corresponding code snippet, allowing effective learning of the respective directive. This significantly improved the quality of directives generated for each parameter. Documentation debt severely inhibits the ability of a single generative model to reliably include these directives. Additionally, our multi-step approach is more capable than a single generative model because each module is adept at handling a specific directive, allowing them to learn task-specific knowledge.

RQ 2

> Detailed docstring generation is ill-modelled as a fully generative task. Owing to an overwhelming amount of documentation debt, the quality of training data is not good enough to enable generative models to learn to include all crucial documentation directives. The difference in the underlying nature of tasks required to generate each directive of the docstring adds to the complexity. Therefore, a combination of multiple models adept at generating each directive is a more reliable approach to generating detailed docstrings.

Future Works The taxonomy proposed by Vidoni and Codabux [55] addresses several other directives of docstrings, like exception raising and data formats, which we do not cover in this work. Further, We need to understand the nature of the tasks each of these directives requires to expand our multi-step model and incorporate suitable steps for each directive. In the future, LLMs should also be explored for their in specific docstring generation tasks. Ensuring that LLMs are actually generating these docstrings rather than

retrieving them is crucial to their utility and is hard to achieve due to the large volumes of data they are trained on. We can also experiment with engineering prompts for LLMs to check for the possibility of achieving higher levels of completeness.

4.3 RQ3. Developers prefer the docstrings from our multi-step approach over those from a single model

We perform a human-centred evaluation of our multi-step approach to accurately evaluate the performance of our approach and verify its superiority over a single model. To this end, we designed an assessment that received an ethical exemption to examine the preference of practitioners of software engineering in parameter docstrings. These practitioners were 17 graduate students in software engineering and majorly using Python in their software development projects. They were recruited through a call sent out to members of a multi-university collaboration effort (names redacted to maintain anonymity). The survey is available as part of our replication package.

We randomly sampled 10 code snippets and their corresponding docstrings generated by three different models: 1) a single model (CodeT5) generating the complete documentation, 2) CodeBERT generating description for the multi-step approach, and 3) CodeT5 generating description for the multi-step approach. We expected each surveyor to give a score between one to six on the Likert scale (i.e., from Poor to Excellent); prior research has demonstrated that six-point Likert scales reduce the uncertainty around the neutral middle point [9]. Our scales assessed the following criteria:

- **Perceived completeness:** The presence of all the expected components of a docstring.
- **Understandability of descriptions:** The ease with which the parameter descriptions can be understood.
- **Grammatical correctness:** The correctness of sentence formations and vocabulary.
- **Technical nature of the content:** The usage of appropriate technical vocabulary.
- **Docstring understandability:** The ease with which the entire docstring can be interpreted.

Table 3: Results obtained from the survey for different criteria of evaluation.

Criteria	Single model (CodeT5)	Multi-step (CodeBERT)	Multi-step (CodeT5)
Perceived completeness	3.310	4.059	4.562
Understandability of descriptions	3.558	4.022	4.457
Grammatical correctness	4.479	4.324	4.524
Technical nature of the content	3.673	4.258	4.544
Docstring understandability	3.566	4.148	4.424

For the parameter descriptions (PD) in our multi-step

approach, we used the outputs from CodeBERT and CodeT5, which were the two best-performing models. The other three directives are produced using the methods d in §3.6. Therefore, the difference in scores between the two multi-step outputs should be interpreted as the difference in parameter descriptions generated by the models. However, the difference in scores between the single-model and multi-step approaches represents a difference in the docstrings as a whole.

From Table 3, we observed that developers found the outputs from the multi-step approaches more complete than those generated by a single-model CodeT5 model. Further, we observed a huge difference in scores for *Understandability of Descriptions* between single-model CodeT5 generating the whole documentation and CodeT5 generating only parameter descriptions in the multi-step approach, which further indicates the overwhelming nature of docstring generation compared to parameter-wise description generation. Additionally, the inclusion of directives, like PT, PN and PV, improved the understandability. The score for *Docstring Understandability* shows variation between single-model and multi-step approaches. There is no significant difference between the scores obtained in the two experiments of the multi-step approach.

The scores for grammatical correctness for all three approaches were comparable, which could be attributed to the models being pre-trained on syntax and semantics of language. The score for the technical nature of the content also shows improvement in the multi-step approach. Overall, developers scored the multi-step approaches higher than the best single-model approach, validating our hypothesis.

CodeT5, as a description generator in the multi-step approach, scores highest in all the cases of evaluation criteria designed for the survey. This is in direct contrast to the scores using automatic metrics in Table 2. This indicates that CodeT5 is more akin to developers' expectations despite CodeBERT having a higher semantic overlap with current documentation standards. Moreover, it is an indication for more human-in-the-loop evaluation metrics to be used for documentation-related tasks to ensure that language models learn to cater to the needs of developers.

Lessons Learned. The results from the survey suggested that the recruited practitioners prefer our multi-step approach due to the inclusion of several docstring directives, improving their completeness. It also contributed to making the docstrings more understandable because the tasks of generating individual directives were simpler than generating an entire docstring. The technical nature of the content is also significantly greater in the multi-step approaches.

> **RQ 3**
>
> Practitioners prefer the output from the multi-step approach with all the technical details over output from a single model with just a description. There is sound acceptance of our approach in terms of completeness and the technical nature of the content.

Future Works. We can expand the scope of our survey

by including developers from different domains of scientific software. Likewise, a Mechanical Turk approach could be used in the future, both as an assessment measure (as is done in this study) and as a data-enhancement technique (i.e., where participants improve the gold data, adding missing directives and mitigating visible cases of documentation debt).

5 IMPLICATIONS

For Researchers. This study provides exploratory insights into docstring documentation for parameters and serves as a baseline study for the inclusion of several technical directives into docstring generation. From discussing the contribution of documentation debt in incomplete documentation to proposing approaches to tackle the issues in training data, we have set a precedent to develop automatic docstring generation tools that mitigate future documentation debt. We add functionality and utility to the established task of code summarization.

Within software engineering, our approach can be further extended to generating directives of returns, exceptions, and other key parts of scientific software documentation. A multi-step approach like ours offers great flexibility and versatility, which can be extended into such directives and to domains beyond software engineering for related tasks.

Our approach also encourages the development of taxonomies for various software use cases to enable taxonomy-guided multi-step approaches like ours and also serve as a representative gold standard for improved evaluation methods. Additionally, investigating the multilingual ability of multi-step models across programming languages could be a future study.

For Developers and Data Scientists. The analysis by Rani et al. [47] describes the need for documentation support required by developers, especially in including relevant information corresponding to crucial directives with automated tools. Generating documentation strongly relies on the training data and, thus, on existing documentation. If developers produce incomplete documentation [1], the trained models will perform similarly.

Therefore, our multi-step approach is an important step towards ensuring docstring completeness and mitigating documentation debt. This is especially relevant for dynamically typed languages, which make understanding code challenging–especially if a single variable can take on multiple datatypes and type-checks have not been coded into specific functions. Our approach is lucrative for developers and scientists looking for effective and extensive documentation generation tools.

6 THREATS TO VALIDITY

We considered several validity threats that arise from the challenges associated with mining software repositories, data preprocessing and multi-step docstring generation.

6.1 Internal Validity Threats

Internal validity threats arise from factors that can alter the outcome [12]. Selection bias while mining scientific software repositories is a potential threat as repositories are mined from GitHub, a corpus containing repositories of all domains and configurations [20]. We mitigated this threat by defining extensive inclusion and exclusion criteria.

Researcher bias during manual annotation is another possible threat while checking for compliance of the retrieved repositories with the inclusion and exclusion criteria, especially for criteria that cannot be automated. To address this bias, two annotators validated the repositories, achieving a respectable 0.779 Cohen's Kappa score [26]. Human bias during preprocessing is another potential risk to reproducibility, which we mitigated by extracting relevant directives from multiple docstring formats, thus maintaining a common format

6.2 External Validity Threats

External validity threats are related to the generalizability of results [12]. In this study, we have considered only Python repositories in the domain of scientific software. Hence, the results can be generalized within the domain. However, we are uncertain about the applicability of the approach and results to non-scientific software repositories or scientific software repositories written in languages other than Python.

6.3 Construct Validity Threats

Human errors and bias could result in inaccuracies in the survey results. We address this threat by handpicking surveyors for the evaluation process, ensuring their qualifications and experience in Python for scientific software.

7 CONCLUSION

In this exploratory study, we aimed to understand the implications of documentation debt in generating detailed docstrings for parameters. We suggest that documenting several docstring directives cannot be achieved by a single generative model. We devise a multi-step approach that uses multiple modules to generate their respective directives, thus enabling each module to learn task-specific information. We conducted a survey of practitioners to rate the documentation generated by our approach and from a single generative model to validate our hypothesis.

There are many potential aspects of the study that would benefit from future research. Firstly, one could expand the task-specific modules to include many other directives of a docstring, like error raising and data formats. One could also study the use of LLMs for generating detailed documentation. Moreover, an evaluation criterion for the whole docstring, adding to directive-specific evaluation, would be of interest.

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In

2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 1199–1210.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. arXiv:1808.01400 [cs.LG]

[3] Vidya M Ayer, Sheila Miguez, and Brian H Toby. 2014. Why scientists should learn to program in Python. *Powder Diffraction* 29, S2 (2014), S48–S64.

[4] 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*. Association for Computational Linguistics, Ann Arbor, Michigan, 65–72. https://aclanthology.org/W05-0909

[5] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[6] Michael Beyene. 2021. *Source-code Summarization of Java Methods Using Control-Flow Graphs*. Ph.D. Dissertation. University of Saskatchewan.

[7] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa Meets Python: A Boa Dataset of Data Science Software in Python Language. In *MSR'19: 16th International Conference on Mining Software Repositories* (Montreal, Canada).

[8] J Burton Browning, Marty Alchin, J Burton Browning, and Marty Alchin. 2014. Docstring Conventions. *Pro Python: Second Edition* (2014), 335–340.

[9] Seung Youn Chyung, Katherine Roberts, Ieva Swanson, and Andrea Hankinson. 2017. Evidence-based survey design: The use of a midpoint on the Likert scale. *Performance Improvement* 56, 10 (2017), 15–23.

[10] Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. *arXiv preprint arXiv:2010.03150* (2020).

[11] Haotian Cui, Chenglong Wang, Junjie Huang, Jeevana Priya Inala, Todd Mytkowicz, Bo Wang, Jianfeng Gao, and Nan Duan. 2022. CodeExp: Explanatory Code Document Generation. *arXiv preprint arXiv:2211.15395* (2022).

[12] Robert Feldt and Ana Magazinius. 2010. Validity threats in empirical software engineering research-an initial survey.. In *Seke*. 374–379.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *CoRR* abs/2002.08155 (2020). arXiv:2002.08155 https://arxiv.org/abs/2002.08155

[14] Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. 2022. A large-scale comparison of Python code in Jupyter notebooks and scripts. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 353–364.

[15] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).

[16] 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. arXiv:2009.08366 [cs.SE]

[17] 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.

[18] 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.

[19] 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)*. Association for Computational Linguistics, Berlin, Germany, 2073–2083. https://doi.org/10.18653/v1/P16-1195

[20] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.

[21] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. arXiv:1902.01954 [cs.SE]

[22] Timothy C Lethbridge, Janice Singer, and Andrew Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE software* 20, 6 (2003), 35–39.

[23] Yang Luo, Wanwangying Ma, Yanhui Li, Zhifei Chen, and Lin Chen. 2018. Recognizing potential runtime types from python docstrings. In *Software Analysis, Testing, and Evolution: 8th International Conference, SATE 2018, Shenzhen, Guangdong, China, November 23–24, 2018, Proceedings 8*. Springer, 68–84.

[24] Paul W McBurney. 2015. Automatic documentation generation via source code summarization. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 903–906.

[25] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

[26] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[27] Yu Meng, Jiaxin Huang, Yu Zhang, and Jiawei Han. 2022. Generating training data with language models: Towards zero-shot language understanding. *Advances in Neural Information Processing Systems* 35 (2022), 462–477.

[28] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2241–2252.

[29] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17 (2012), 703–737.

[30] Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. 2010. A survey of scientific software development. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*. 1–10.

[31] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. arXiv:2205.11739 [cs.SE]

[32] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. *arXiv preprint arXiv:2302.04026* (2023).

[33] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. arXiv:2201.01549 [cs.SE]

[34] Travis E Oliphant. 2007. Python for scientific computing. *Computing in science & engineering* 9, 3 (2007), 10–20.

[35] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023).

[36] 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*. Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[37] Aleksandra Pawlik, Judith Segal, and Marian Petre. 2012. Documentation practices in scientific software development. In *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*. IEEE, 113–119.

[38] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. https://doi.org/10.1145/3510003.3510038

[39] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. arXiv:2105.08645 [cs.AI]

[40] Gustavo Pinto, Igor Wiese, and Luiz Felipe Dias. 2018. How do scientists develop scientific software? An external replication. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 582–591.

[41] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220.

[42] Roger S Pressman. 2005. *Software engineering: a practitioner's approach*. Palgrave macmillan.

[43] Weizhen Qi, Yeyun Gong, Yu Yan, Can Xu, Bolun Yao, Bartuer Zhou, Biao Cheng, Daxin Jiang, Jiusheng Chen, Ruofei Zhang, Houqiang Li, and Nan Duan. 2021. ProphetNet-X: Large-Scale Pre-training Models for English, Chinese, Multi-lingual, Dialog, and Code Generation. arXiv:2104.08006 [cs.CL]

[44] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[45] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[46] Pooja Rani, Suada Abukar, Nataliia Stulova, Alexandre Bergel, and Oscar Nierstrasz. 2021. Do comments follow commenting conventions? a case study in java and python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 165–169.

[47] Pooja Rani, Mathias Birrer, Sebastiano Panichella, Mohammad Ghafari, and Oscar Nierstrasz. 2021. What do developers discuss about code comments?. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 153–164.

[48] Judith Segal. 2007. Some problems of professional end user developers. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE, 111–118.

[49] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. 2015. To document or not to document? An exploratory study on developers' motivation to document code. In *Advanced Information Systems Engineering Workshops: CAiSE 2015 International Workshops, Stockholm, Sweden, June 8-9, 2015, Proceedings 27*. Springer, 100–106.

[50] Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. *Information processing & management* 45, 4 (2009), 427–437.

[51] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 71–80.

[52] Matúš Sulír and Jaroslav Porubän. 2017. Generating method documentation using concrete values from executions. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[53] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. 2017. Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method. *Information and Software Technology* 85 (2017), 43–59. https://doi.org/10.1016/j.infsof.2017.01.006

[54] Melina Vidoni. 2022. Understanding roxygen package documentation in R. *Journal of Systems and Software* 188 (2022), 111265.

[55] Melina Vidoni and Zadia Codabux. 2023. Towards a Taxonomy of Roxygen Documentation in R Packages. *Empirical Software Engineering* XX (2023), XX.

[56] Haifeng Wang, Jiwei Li, Hua Wu, Eduard Hovy, and Yu Sun. 2022. Pre-trained language models and their applications. *Engineering* (2022).

[57] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *CoRR* abs/2109.00859 (2021). arXiv:2109.00859 https://arxiv.org/abs/2109.00859

[58] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. 2018. CBAM: Convolutional Block Attention Module. arXiv:1807.06521 [cs.CV]

[59] Haiyin Zhang, Luís Cruz, and Arie Van Deursen. 2022. Code smells for machine learning applications. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. 217–228.

[60] Wenhao Zheng, Hongyu Zhou, Ming Li, and Jianxin Wu. 2019. CodeAttention: translating source code to comments by exploiting the code constructs. *Frontiers of Computer Science* 13 (2019), 565–578.