Università
della
Svizzera
italiana

Faculty
of Informatics

Dissertation Proposal

Wednesday 21st December, 2022

Automating Code-Related Tasks via Pre-trained Models of Code

Injecting Log Statements and Summarizing Code Snippets

Antonio Mastropaolo

Abstract

Developers are often faced with the challenge of writing high-quality code while meeting strong time constraints. Recent literature exploits Deep Learning (DL) models to support developers in code-related tasks. For example, DL-based approaches have been proposed to automate bug-fixing activities, code summarization, and code review. Some of these tasks require to work with both code and technical natural language (*e.g.,* code summarization), posing additional challenges in the training of DL models which must deal with bi-modal data. Our goal is to widen the support given to developers when dealing with code-related tasks characterized by technical natural language and code. To this extent, we started investigating the benefits brought by the "pretrain-then-finetune" paradigm when using DL models to automate code-related activities. The basic idea of this paradigm is to first pre-train the model with self-supervised tasks with the only goal of learning the languages of interest (*e.g.,* technical English and code). Then, the fine-tuning phase takes care of specializing the model for the specific task of interest (*e.g.,* code summarization). Given the positive results we achieved, we focused our research on two code-related tasks characterized by both code and natural language. The first is the already mentioned code summarization which consists in generating a code summary for a given piece of code at hand (*e.g.,* method, code snippet). The second is the generation and injection of complete log statements in which the DL model takes as input a code component and is in charge of recommending to developers which log statements may be beneficial to inject, thus taking care of generating the log statement (including a meaningful log message) and inject it in the correct code location.

Research Advisor
Prof. Gabriele Bavota

Internal Committee Members
Prof. Antonio Carzaniga, Prof. Silvia Santini

External Committee Members
Prof. Davide Di Ruscio, Dr. Alexey Svyatkovskiy

This proposal has been approved by the dissertation committee and the Ph.D. program directors:

Prof. Gabriele Bavota, Research Advisor and Committee Chair
Università della Svizzera italiana (USI), Switzerland

Prof. Antonio Carzaniga, Internal Committee Member
Università della Svizzera italiana (USI), Switzerland

Prof. Silvia Santini, Internal Committee Member
Università della Svizzera italiana (USI), Switzerland

Prof. Davide Di Ruscio, External Committee Member
University of L'Aquila, Italy

Dr. Alexey Svyatkovskiy, External Committee Member
Microsoft Research, USA

Prof. Walter Binder, Ph.D. Program Director
Università della Svizzera italiana (USI), Switzerland

Prof. Stefan Wolf, Ph.D. Program Director
Università della Svizzera italiana (USI), Switzerland

Contents

Contents iii

1 Introduction 1
1.1 Expected Contributions and State of the Research 2
1.2 Other Exploratory Studies 4

2 Background and State of the art 5
2.1 Automating Logging Activities 5
2.2 Code Summarization 6

3 Studying the Usage of Pre-trained Language Models to Support Code-Related Tasks 9
3.1 Studying the Usage of Text-to-Text Transfer Transformer to Support Code-Related Tasks [1] 9
3.2 Using Transfer Learning for Code-Related Tasks [2] 11
3.3 Replication Package 12

4 Automating Logging Activities 13
4.1 Using DL to Generate Complete Log Statements 13
4.2 Future Directions 16
4.3 Replication Package 16

5 Code Summarization 17
5.1 An Empirical Study on Code Comment Completion 17
5.2 Future Directions 18

6 Conclusions and Future Plans 20

Chapter 1

Introduction

The ever-increasing complexity of software systems often pushes developers to look for a helping hand either from a team-mate or, when not possible, by relying on (semi-)automated tools. In response to these needs, researchers have proposed recommender systems for software developers. Robillard *et al.* [3] defined such systems as "*software tools that can assist developers with a wide range of activities, from reusing code to writing effective bug reports*". The first generation of recommender systems for software developers was characterized by approaches built on top of limited and manually-crafted heuristics. For instance, Moreno *et al.* [4] presented *JSummarizer*, an eclipse plug-in to automatically generate a natural language description of a Java class by using a set of pre-defined templates. Although empirical studies have shown the potential usefulness of these tools [5, 6, 7], they suffer of generalizability issues. For example, the number of templates that can be manually defined to document Java classes is clearly limited, and unlikely to cover all possible coding scenarios.

Lately, the surge of software development data hosted on platforms such as GitHub, paved the way to a new class of approaches based on *data-driven* recommenders that can learn how to automate code-related tasks by "looking" at activities performed by real developers in open source projects. To give the reader a better idea of the quantity of software data that can be found on these code repositories, as of June 2022, GitHub counts more than 86 Million developers and more than 200 Million repositories [8]. Such a sheer amount of data unlocked the usage of deep learning (DL) models to support code-related tasks, such as automatic bug-fixing [9, 10, 11, 12], code summarization [13, 14, 15] and code completion [16, 17, 18, 19, 20, 21]. To better understand how such techniques work, let us discuss the DL model proposed by Tufano *et al.* [9] to automatically fix bugs in Java methods. The model learned bug-fixing patterns by being trained on ∼58k pairs of buggy and fixed methods mined from software repositories. In particular, these are methods which have been subject to bug-fixing activities and, as such, it is possible to retrieve from software repositories their version before (buggy) and after (fixed) the bug-fix. Despite the large-scale training, the DL model was able to fix only ∼9% of bugs in Java methods as a real developer would do, posing questions about the possibility of integrating such an approach into recommender systems that can support developers during bug-fixing activities.

When considering tasks that involve both natural language and programming languages (*e.g.,* code summarization), results achieved by DL-based techniques proposed in the literature are even further from what would be considered acceptable to be deployed in practice.

For instance, Haque *et al.* [14] presented a DL-based model that can correctly generate a code summary as the original one written by the developer in only ∼3% of cases. Thus, despite the progress in this field, the support that state-of-the-art techniques can offer to developers is still limited, pointing to the need for more research.

1.1 Expected Contributions and State of the Research

The *goal* of my Ph.D. is to investigate the usage of pre-trained DL models to support code-related tasks, with a specific focus on tasks characterized by the presence of bi-modal data in the form of technical natural language (*e.g.,* code comments) and source code. With "pre-trained DL models" we refer to models which are pre-trained with self-supervised tasks to learn characteristics of the language of interest (*e.g.,* Java). These models can then be fine-tuned in a supervised way to learn how to automate specific tasks (*e.g.,* code summarization).

The idea of using the "pretrain-then-finetune" paradigm to support software engineering (SE) tasks has been firstly proposed by Robbes *et al.* [22], which suggested it as a way to overcome the limited size of training datasets available for specific tasks (*e.g.,* sentiment analysis on software-related corpora such as Stack Overflow discussions). Following, the advent of the Transformer DL architecture [23] substantially accelerated the adoption of pre-trained models in SE. Concrete examples of these applications are *CodeBERT* [24], a pre-trained model for programming language and natural language, and *IntelliCode Compose* [16], an advanced code completion technique exploiting a multi-layer generative pretrained Transformer for code (GPT-C).

Building on top of this literature, we started by empirically investigating the performance of pretrained models on several code-related tasks [1], namely *bug-fixing*, *mutants injection*, *asserts generation*, and *code summarization*. We focused on the *Text-to-Text-Transfer-Transformer* model (T5) [25], a transformer-based model which represents any task as a text-to-text transformation (*i.e.,* text is provided as input and text is generated as output). T5 also allows *multi-task* learning, meaning that a single model can be trained to learn several tasks at the same time. We pre-trained T5 using the classic *masked language model* task, consisting in randomly masking 15% of the tokens composing the input sequence, asking the model to predict those tokens that have been masked. In our case, the input sentences were composed by Java code and natural language, since those are the two languages involved in the four tasks we targeted (Java code in all four of them, natural language in *code summarization*). We then fine-tuned T5 using a multi-task setting (*i.e.,* a single model has been trained to solve all four above-described tasks). For each of the experimented tasks, we compared the achieved performance with the state-of-the-art techniques [9, 26, 27, 14], showing that T5 was able to improve the performance of the competitive approaches in all four tasks when pre-trained on bi-modal data (*i.e.,* natural language and code) and fine-tuned in a multi-task setting. This approach and its evaluation have been accepted at ICSE 2021 [1]:

> **Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks**
> Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Roco Oliveto, Gabriele Bavota. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE 2021)*, pp. 336-347.

While the results achieved in this paper looked promising, two important questions were left unanswered. First, we did not investigate the role played by pre-training the model and then fine-tuning it.

In other words, we were not aware of the benefits (if any) brought by pre-training the model, since we did not compare a pre-trained T5 with a non pre-trained T5 (*i.e.,* only fine-tuning). Second, we did not study the impact on performances of mixing tasks (*i.e.,* multi-task) when specializing the model on the set of code-related tasks we aimed at supporting.

For such reasons, we extended our ICSE 2021 work by investigating these two specific aspects. We reported the results of this second investigation in the following paper [2]:

> **Using Transfer-Learning for Code-Related Tasks**
> Antonio Mastropaolo, Nathan Cooper, David Nader-Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, Gabriele Bavota. In *IEEE Transactions on Software Engineering (TSE 2022)*, to appear.

Our findings showed the boost in performance provided by the pretraining phase, while did not highlight a major role of the multi-task learning.

Once established the effectiveness of the "*pretrain-then-finetune*" paradigm in dealing with code-related tasks, we decided to explicitly focus on two tasks which are characterized by a mix of code and natural language and for which support in the literature is lacking: (i) code snippet summarization (*i.e.,* the ability to describe in natural language a snippet of code), and (ii) generation and injection of complete log statements.

As for the former (*i.e.,* code snippet summarization), the results of our ICSE 2021 paper [1] showed that even the pretrained T5 struggles in generating summaries for entire Java methods (∼10% correct predictions). Thus, as a first step, we studied whether by simplifying the problem it was possible to achieve better results. In particular, we moved from the automated generation of complete summaries, to code comment completion, in which the "machine" is in charge of completing a comment that the developer starts writing, similarly to what has been already done for code tokens by code completion techniques [18, 16, 17, 20, 21, 19]. To this extent, we instantiated the T5 model on the code comment completion task. Such a study has been accepted for publication at ICSME 2021 [28]:

> **An Empirical Study on Code Comment Completion**
> Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, Gabriele Bavota. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME)*, pp. 159-170.

Despite the better results achieved for this simpler task as compared to the more challenging code summarization, the achieved performance were still far from what developers would consider useful to deploy in recommender systems (∼16% correct predictions). This explains our shift towards *code snippet* summarization: While entire methods could be too challenging to describe for a DL model, focusing on shorter code snippets composed by a few statements may be simpler and still properly support program comprehension activities. A first challenge to overcome here is the building of the training dataset. Indeed, while it is easy to create datasets composed of pairs <method, javadoc> to train a DL model supporting method-level summarization, this is not the case for code snippets. Thus, we started by building a manually-labeled dataset of $\langle CodeSnippet, CodeComment \rangle$ that can be used to train DL models for the task of snippet-level summarization. As of Today, our dataset counts over 11k pairs and will be soon made available to the research community. Our next steps on this topic will target the training and evaluation of models models supporting snippet-level code summarization.

As for the second problem we target (*i.e.,* generation of complete log statements), we aim at providing complete support to developers while performing logging activities. Indeed, while the benefits of logging are undisputed, making appropriate decisions about where to inject log statements, what information to log, and at which log level (*e.g.,* error, warning) is crucial for the logging effectiveness, besides being time-consuming and challenging for developers. Towards this goal, we presented LANCE (Log stAtemeNt reCommEnder) [29], the first approach in the literature able to generate complete log statements (including a meaningful natural language message) and inject them in Java code. The approach and its evaluation have been accepted for publication at ICSE 2022 [29]:

> **Using Deep Learning to Generate Complete Log Statements**
>
> Antonio Mastropaolo, Luca Pascarella, Gabriele Bavota. In *Proceedings of the 44rd International Conference on Software Engineering (ICSE 2022)*, pp. 2279-2290.

As the next step in this direction, we are working towards addressing some of LANCE's limitations. Indeed, LANCE always assumes the need for one log statement in a Java method provided as input. We want to address this issue training LANCE in order to decide a proper number of statements which may be needed (0 to n). Also, we want to boost its performance especially in the generation of meaningful natural language log messages.

1.2 Other Exploratory Studies

Before defining the path of the research presented in this proposal, we also investigated the extent to which *"data-driven techniques"* can support the task of automated variable renaming in the context of Java methods. To conduct the study, we empirically evaluated the performances of three different approaches: (i) an n-gram cached language model [30]; (ii) the T5 model [25]; and (iii) the Transformer-based model presented by Liu *et al.* [19]. This work has recently been accepted for publication in the Journal of Empirical Software Engineering (EMSE).

Chapter 2

Background and State of the art

Given the focus of my research, I will discuss the state-of-the-art concerning (i) the automation of logging activities, and (ii) techniques supporting source code summarization.

2.1 Automating Logging Activities

As previously mentioned, developers must take several decisions when logging their code (*i.e.,* logging location, level, message). To support in these activities, the research community presented several (semi-)automated techniques that we discuss in the following based on the specific type of support they provide.

Log message enhancement. Yuan *et al.* [31] proposed LogEnhancer as a prototype to automatically recommend relevant variable values for each log statement, refactoring its message to include such values. Their evaluation on eight systems shows that LogEnhancer can dramatically reduce the set of potential root failure causes when inspecting log messages. Liu *et al.* [32] tackled the same problem using, however, a customized deep learning network. Their evaluation showed that the mean average precision of their approach is over 84%.

Log placement. Other researchers targeted the suggestion of the code location for log statements [33, 34, 35]. Zhu *et al.* [36] presented LogAdvisor, an approach to recommend where to add log statements. The evaluation of LogAdvisor on two Microsoft systems and two open-source projects resulted in an accuracy of 60% when applied on pieces of code lacking log statements. Yao *et al.* [37] tackled the same problem in the specific context of monitoring the CPU usage of web-based systems. The proposed tool, named Log4Perf, identifies code blocks to log as those associated with web requests having unstable performances.

Li *et al.* [38] proposed a deep learning framework to recommend logging locations at the code block level. They report a 80% accuracy in suggesting logging locations using within-project training, with slightly worst results (67%) in a cross-project setting. Cândido *et al.* [39] investigated the effectiveness of log placement techniques in an industrial context, showing that models trained on open source code can be effectively used in industry.

Log level recommendation. A third family of techniques focus on recommending the proper log level (*e.g.,* error, warning, info) for a given log statement [40, 41]. Mizouchi *et al.* [42] proposed PADLA as an extension for Apache Log4j framework to automatically change the log level for better record of runtime information in case of anomalies. The DeepLV approach proposed by Li *et al.* [43] uses instead a deep learning model to recommend the level of existing log statements in methods. DeepLV aggregates syntactic and semantic information of the source code and showed its superiority with respect to the state-of-the-art.

The evaluation, that has been conducted on nine large-scale open source projects, highlighted that using exclusively the syntactic context of the log statement is sufficient to identify wrong log levels used by developers in most of cases, whereas relying only on the log message (*i.e.,* semantic information) can be useful for specific types of log level (*i.e.,* info, error).

Summing Up

While several approaches have been proposed to partially automate logging activities [40, 44, 45, 46, 47, 48, 49, 50], none of them can provide a comprehensive support to developers for such a task. Indeed, they all focus on specific sub-tasks such as log placement or log level prediction, not targeting the complete generation and placement of complete log statements. In our ICSE 2022 paper we presented LANCE [29], the first approach providing complete support for synthesizing and injecting complete log statements in Java code, thus addressing the three previously described sub-tasks, namely: generation of a meaningful log message, correct placement of the log statement in a given code, and selection of an appropriate log level. LANCE is detailed in Chapter 5.

2.2 Code Summarization

Several techniques have been proposed to automatically summarize source code [51, 52, 53, 54, 55, 14, 56, 57, 58]. Those can be roughly classified into two categories: extractive [51, 52, 53, 54, 55] and abstractive [14, 56, 57, 58]. The former create a summary of a code component which includes information extracted from the component being summarized (*e.g.,* a bag of words or just a selection of code statements considered particularly relevant for the comprehension), while the latter may include in the generated summaries information that is not present in the code component to document.

The evaluation of these approaches is done using quantitative and qualitative measures. An example of quantitative metrics is the number of correct predictions, namely cases in which the generated summary is identical to the reference one, usually written by developers. Such a metric has however a strong drawback: If the generated summary is different from the target one but semantically equivalent (*i.e.,* it conveys the same message by using a different wording), it is considered wrong. To partially address this issue, metrics from the NLP field are also adopted, such as BLEU [59] and METEOR [60]. Both these metrics produce as output a score ranging between 0 (the sequences are entirely different) and 1 (the sequences are identical). Such a score is computed by looking at overlapping n-grams between the generated and the reference summary. The qualitative analysis, instead, usually involves a manual inspection of the generated summaries with the goal of assessing their relevance and meaningfulness for the code to summarize.

Both extractive and abstractive techniques have been proposed to document code components at different granularity levels, such as method (*e.g.,* [51, 55, 61, 62, 52, 63]), method parameters (*e.g.,* [52]), method usages (*e.g.,* [57]), class (*e.g.,* [64, 65, 53]), unit tests [66], and code snippets (*e.g.,* [56, 67]).

Given our focus to automatically document code snippets using pre-trained DL models, we focus our discussion on (i) techniques aimed at documenting code snippets (regardless of the underlying technique) and (ii) DL-based approaches (regardless of the target granularity).

Most of the proposed DL-based code summarization techniques work at method-level granularity.

Such a design choice is dictated by the availability of training data: For example, in the case of Java, it is easy to create datasets composed of pairs <method, javadoc> that can be fed to DL models to "teach" them how to summarize a method. Such a comment-to-code linking is instead non-trivial when it comes to inner comments (*i.e.,* comments within a method) documenting a few statements, which is the focus of our work.

Hu *et al.* [58] were the first to propose the use a Deep Neural Network (DNN) to automatically generate comments for a given Java method. The authors mine ∼9k Java projects hosted on GitHub to collect pairs of <method, comment>, where "comment" is the first sentence of the Javadoc linked to the method. These pairs, properly processed, are used to train and test the DNN. The authors assess the effectiveness of their technique by using the BLEU-4 score [59], showing the superiority of their approach with respect to the competitive technique presented in [68] and not exploiting DL.

While Hu *et al.* represent the code to document as a stream of tokens without capturing the structural representation of code, LeClair *et al.* [13] show that combining the AST source code structure with a token-based representation can help DL-based code summarization. Their approach, tested on 2.1M methods, showed its superiority as compared to the previous works by Hu *et al.* [58] and Iyer *et al.* [68].

Also Haque *et al.* [14] focus on the documentation of Java methods through an encoder-decoder architecture [69] but, in this case, three inputs are provided to the model to generate the summary: (i) the source code of the method, as a flattened sequence of tokens representing the method (similarly to Hu *et al.* [58]); (ii) its AST representation (similarly to LeClair *et al.* [13]); and (iii) the "file context", meaning the code of every other method in the same file. The authors show that adding the contextual information as one of the inputs substantially improves the BLEU score obtained by DL techniques.

Subsequent works [70, 71] investigated the usage of the transformer architecture [23] to support the task of method-level code summarization. Wu *et al.* [71] proposed a variation of the common transformer architecture capable of obtaining new state-of-the-art results as compared to the baselines [68, 72, 73, 58, 70].

Hybrid techniques have also been proposed. Wan *et al.* [73] combined a NMT (Neural Machine Translation) model [69] with reinforcement learning to summarize python functions. The authors obtained better performance as compared to several DL-based architectures, such as Seq2Seq [74], Seq2Seq+Att [69], Tree2Seq [75], Tree2Seq+Attn [72].

Besides approaches that can produce summaries at method-level granularity, researchers have also presented techniques capable of recommending summaries for code snippets, thus working at a lower granularity level. To this end, most of the techniques targeting the documentation of code snippets are based on IR (Information Retrieval). Representative works in this area are CodeInsight [76], ColCom [77], [78] and ADANA [79].

Rahman *et al.* [76] proposed CodeInsight, a technique that retrieves from Stack Overflow sentences which are relevant for a given code segment and can be used to document possible deficiencies it has. The evaluation has been performed on 292 Stack Overflow code snippets and 5,039 discussion comments, reporting a recall of 85.42% and a MRR of 0.44, on average. Moreover, a user study involving professional developers showed that 80% of the comments recommended by CodeInsight are considered relevant by practitioners.

Wong *et al.* [77] presented ColCom, an approach to identify in software repositories code comments that can be reused to document a given snippet. The authors report that 23.7% of the retrieved comments could be used to document the code snippet, showing that the overall accuracy of the technique was relatively low.

The most recent IR-based approach for code snippet documentation is ADANA, proposed by Aghajani *et al.* [79] to document code snippets in Android apps. ADANA reuses the descriptions of similar code snippets retrieved from GitHub Gist and Stack Overflow. The authors show that the comments retrieved by ADANA can help in comprehending code.

Another family of techniques related to code snippets documentation relies on manually defined templates to describe high-level actions performed within functions. Seminal work in this area are from Sridhara *et al.* [56] and Wang *et al.* [80]. These approaches, while valuable, cannot generalize to all combinations of code statements one could expect to find since they are based on predefined templates. For this reason, data-driven techniques exploiting DL have been proposed [67, 81].

When it comes to snippet-level granularity, RL-BlockCom [67] represents the state-of-the-art. RL-BlockCom is a composite DL model combining reinforcement learning with a classic encoder-decoder model. With such a novel framework, the authors show that RL-BlockCom can achieve a BLEU-4 of 24.28, outperforming previous state-of-the-art models [82, 69].

Summing Up

Although several approaches have been presented to automatically document classes or methods, little effort has been devoted to more fine-grained documentation (*e.g.,* documenting code snippets or even a single statement). This is mostly due to the difficulty in building high-quality training sets pairing code snippets to their description. While RL-BlockCom [67] represents a first step in the direction of DL-based snippet summarization, it suffers of some major limitations mostly related to the way in which its training/test sets have been built:

1. Simplified/unrealistic linking of code comment to the documented snippet. This is due to some of the design choices made by the authors. For example, they "*regard the first out-of-scope statement as the demarcation point of the scope of the comment*". This means that, accordingly to their approach, it is not possible for a code comment to document non-contiguous statements. As we will show, the dataset we are building which currently counts 11,563 inner comments linked to the documented code snippets indicates that ∼28% of them (3,250) document non-contiguous statements. These are all cases which cannot be successfully supported by *RL-BlockCom*.

2. Lack of filters to identify code summaries. Chen *et al.* correctly observed that not all comments "describe" code statements. Thus, they use heuristics to remove commented out code, TODO comments, IDE-generated comments, and non-text comments containing dates or links. Despite these filters, using such an approach to create a training dataset for a snippet summarization approach such as *RL-BlockCom* means feeding it with comments which may not be an actual code summary of the documented snippet. For example, when manually looking at the previously mentioned 11,563 instances, we found 40% of them to just act as a logical split of source code (*i.e.,* a "formatting" comment [83]) without providing additional information on the documented code (*e.g.,* a comment `//get messages` put on top of a method call `getMessages()`). These comments are useless to train a code summarizer, but are not excluded from the *RL-BlockCom* training dataset.

3. The training dataset used in RL-BlockCom includes code summaries as short as two words [67]. These are unlikely to be code summaries useful to support program comprehension.

Our goal is to (i) contribute a manually curated dataset featuring thousands of documented code snippets; and (ii) using it to achieve new state-of-the-art results in the context of snippet-level code summarization thanks to the usage of pre-trained Transformers.

Chapter 3

Studying the Usage of Pre-trained Language Models to Support Code-Related Tasks

As first step in our research we investigated the effectiveness of pre-trained language models when applied to a variety of code-related tasks, including those characterized by bi-modal data (*i.e.,* involving both source code and technical natural language). Such a work has been presented at ICSE 2021 [1] and later on extended on TSE [2]. We summarize these two works in the following sections.

3.1 Studying the Usage of Text-to-Text Transfer Transformer to Support Code-Related Tasks [1]

Recent years have seen the raise of transfer learning in the field of NLP. In one of its variation, the basic idea is to first pre-train a model on a large and generic dataset by using a self-supervised task (*e.g.,* masking tokens in strings and asking the model to guess the masked tokens). Then, the trained model is fine-tuned on smaller and specialized datasets, each one aimed at supporting a specific task. In this context, Raffel *et al.* [25] proposed the T5 model, pre-trained on a large natural language corpus and fine-tuned to achieve state-of-the-art performance on many tasks, all characterized by text-to-text transformations.

We empirically investigated the potential of a T5 model when pre-trained and fine-tuned to support several code-related tasks. We started by pre-training a T5 model using a dataset featuring bi-modal data including 336,548 English sentences and 2,346,018 source code components (*i.e.,* Java methods). Then, we fine-tune the model using four datasets from previous work with the goal of supporting four code-related tasks:

Automatic bug-fixing. We use the dataset by Tufano *et al.* [9], composed of instances in which the "input string" is represented by a buggy Java method and the "output string" is the fixed version of the same method. Note that for this task the authors applied an abstraction process to the code to reduce the size of its vocabulary. This means, for example, that all variable identifiers are replaced with a special token "VAR_ID", where ID is a progressive number indicating whether the variable is the first, the second, *etc.* defined in the Java method. The abstraction process includes a mechanism to transform back the abstracted code into its raw format.

Table 3.1. Correct predictions for T5 and the Baselines

Task	Dataset	T5	Baseline
Bug-Fixing [9]	BF_{small}	**10%**	9%
	BF_{medium}	3%	3%
Assert-Generation [27]	AG_{raw}	**47%**	18%
	AG_{abs}	**34%**	31%
Mutant Generation [26]	MG_{ident}	**28%**	17%
Code Summarization [14]	CS	**11%**	3%

Thus, even if the DL model predicts the abstracted version of the "fixed method", this can be converted back to raw source code to be recommended to the developer.

Injection of code mutants. This dataset is also by Tufano *et al.* [26], and features instances in which the input-output strings are reversed as compared to automatic bug-fixing (*i.e.,* the input is a fixed method, the output is its buggy version). The model must learn how to inject bugs (mutants) in code instead of fixing bugs. Also for this task code abstraction is used.

Generation of assert statements in test methods. We use the dataset by Watson *et al.* [27], composed of instances in which the input string is a representation of a test method without an assert statement and a focal method it tests (*i.e.,* the main production method tested), while the output string encodes an appropriate assert statement for the input test method.

Code Summarization. We use the dataset by Haque *et al.* [14] where input strings are representations of a Java method to summarize and the output string is a textual summary.

The choice of the four tasks subject of our study (*i.e., bug-fixing*, *mutants injection*, *asserts generation*, and *code summarization*) is dictated by the will of experimenting with tasks that use, represent, and manipulate code in different ways. In particular, we include in our study tasks aimed at (i) transforming the input code with the goal of changing its behavior (*bug-fixing* and *mutants injection*); (ii) "comprehending code" to verify its behavior (*asserts generation*); and (iii) "comprehending code" to summarize it in natural language (*code summarization*).

Once the T5 model has been fine-tuned on all these tasks, we run it on the same test sets used in the four referenced works [9, 26, 27, 14] comparing the achieved results to those reported in the original work. Table 3.1 presents the achieved results by comparing T5 and the state-of-the-art baselines in terms the percentage of correct predictions, namely cases in which the output generated by the experimented technique is identical to the expected one (*e.g.,* in the case of *bug-fixing* the approach fixed the bug exactly as real developers did; similarly, in the case of *code summarization* the textual summary generated by the approach for a given Java method is identical to the one written by developers in it Javadoc documentation). Note that for some tasks multiple datasets were available in the original works presenting the baselines, and all of them have been used in our comparison (*e.g.,* for *bug-fixing*, Tufano *et al.* created a dataset BF_{small} composed by Java methods up to 50 tokens and BF_{medium} composed by Java methods up to 100 tokens).

The achieved results (see Table 3.1) clearly show the gain in performance ensured by T5 over the state-of-the-art baselines. The gap is limited for the task of bug-fixing, while substantial when looking at the other three tasks. For example, in the case of *code summarization*, namely the only task employing bi-modal data, the pre-training performed on both English sentences and Java source code seems to provide a substantial boost in performance as compared to the DL-based baseline by Haque *et al.* [14] (11% *vs* 3% of correct predictions).

Additional analyses including qualitative examples of predictions generated by the experimented models are available in the ICSE'21 paper [1].

It is worth noticing that in this work we fine-tuned a single pre-trained T5 model in a multi-task setting on all four tasks, showing that it is able to achieve better results as compared to the four referenced baselines. However, since we only experimented with a pre-trained model fine-tuned in a multi-task setting, questions about the actual advantage offered by transfer learning remained unanswered. We addressed such limitations in the follow-up work discussed in Section 3.2.

3.2 Using Transfer Learning for Code-Related Tasks [2]

We address the limitations of our ICSE'21 paper by investigating the extent to which *transfer learning* is beneficial when dealing with code-related tasks. We studied the impact on performance of both pre-training and multi-task learning. The *context* is represented by the datasets introduced in Section 3.1, (*i.e.,* the ones by Tufano *et al.* for bug fixing [9] and injection of mutants [26], by Watson *et al.* for assert statement generation [27], and by Haque *et al.* for code summarization [14]). We assess the performance of T5 in the following scenarios:

- **No Pre-training:** We do not run any pre-training step. We directly fine-tune four different T5 models, each one supporting one of the four tasks we experiment with.

- **Pre-training single task:** We first pre-train the T5 model on the dataset described in Section 3.1. Then, starting from it, we fine-tune four models, one for each single task.

- **Pre-training Multi-Task**: Lastly, we fine-tune the pre-trained model using a multi-task learning framework in which we train a single model to support all four code-related tasks. We experiment with two different multi-task fine-tunings: (i) the first is the one used in our original paper [1], in which the percentage of training instances from each of the four tasks is proportional to the size of their training dataset, (ii) the second in which the percentage of training instances is the same for all four tasks (*i.e.,* 25% per task).

The achieved results showed that: (i) the pre-training process of the T5, as expected, boosts its performance across all tasks; (ii) the multi-task fine-tuning (*i.e.,* a single model trained for different tasks) instead, does not consistently help in improving performance, possibly due to the different types of "data" manipulated in the four tasks (*i.e.,* raw code, abstracted code, natural language); (iii) in its best configuration, we confirm that T5 performs better that the baselines across all four tasks.

To conclude this chapter, we discuss qualitative examples of predictions generated by T5 to give a better idea to the reader about the capabilities of this model in supporting the four code-related tasks. Fig. 3.1 shows two examples of correct predictions made by T5 for each task. Each example is bordered with a dashed line and shows (i) the input provided by the model, and (ii) the generated output. In particular, in the case of the bug-fixing, mutants injection, and code summarization tasks, the first line shows the input and the second the output. Concerning the generation of assert statements, the first two lines (*i.e.,* those marked with "//Test method" and "//Focal method") represent the input, while the third line shows the generated assert statement. We highlighted in bold the most relevant parts of the output generated by the model.

Figure 3.1. Examples of perfect and alternative predictions

Bug-fixing

```
private void METHOD_1 ( int VAR_1 , int offset ) { switch ( VAR_1 ) { case 0 : this . VAR_2 = offset ; case 1 : this . VAR_3 = offset ; case INT_1 :
this . VAR_4 = offset ; case INT_2 : this . VAR_5 = offset ; } }

private void METHOD_1 ( int VAR_1 , int offset ) { switch ( VAR_1 ) { case 0 : this . VAR_2 = offset ; break ; case 1 : this . VAR_3 = offset ; break ;
case INT_1 : this . VAR_4 = offset ; break ; case INT_2 : this . VAR_5 = offset ; break ; } }
```

```
public void METHOD_1 ( final long [ ] data , boolean length ) { int VAR_1 = ( data . length ) * ( VAR_2 ) ; if ( length ) METHOD_2 ( VAR_1 ) ; METHOD_3
( ( ( position ) + VAR_1 ) ) ; VAR_3 . METHOD_4 ( data , VAR_4 , null , ( ( VAR_5 ) + ( position ) ) , VAR_1 ) ; position += VAR_1 ;
end = ( ( position ) > ( end ) ) ? position : end ; }

public void METHOD_1 ( final long [ ] data , boolean length ) { int VAR_1 = data . length ; if ( length ) METHOD_2 ( VAR_1 ) ; VAR_1 *= VAR_2 ;
METHOD_3 ( ( ( position ) + VAR_1 ) ) ; VAR_3 . METHOD_4 ( data , VAR_4 , null , ( ( VAR_5 ) + ( position ) ) , VAR_1 ) ; position += VAR_1 ;
end = ( ( position ) > ( end ) ) ? position : end ; }
```

Mutants generation

```
public int METHOD_1 ( int a ) { int index = VAR_1 [ ( a + 1 ) ] ; return VAR_2 . METHOD_2 ( index ) ; }

public int METHOD_1 ( int a ) { int index = VAR_1 [ a ] ; return VAR_2 . METHOD_2 ( index ) ; }
```

```
public java.lang.String METHOD_1 ( final java.lang.String VAR_1 ) { return STRING_1 . METHOD_2 ( VAR_2 . METHOD_3 ( ) . METHOD_4 ( ) ) ; }

public java.lang.String METHOD_1 ( final java.lang.String VAR_1 ) { return VAR_2 . METHOD_3 ( ) . METHOD_4 ( ) ; }
```

Generation of assert statements

```
// Test method
shouldNeverValidateNullUserIV ( ) { final uk . gov . gchq . gaffer . federatedstore . FederatedAccess access = new uk . gov . gchq . gaffer .
federatedstore . FederatedAccess . Builder ( ) . addingUserId ( null ) . build ( ) ; "<AssertPlaceHolder>" ; }
// Focal method
isValidToExecute ( uk . gov . gchq . gaffer . user . User ) { return ( isPublic ) || ( ( null != user ) && ( ( isAddingUser ( user ) )
|| ( ( ! ( isAuthsNullOrEmpty ( ) ) ) && ( isUserHasASharedAuth ( user ) ) ) ) ) ; }

org . junit . Assert . assertFalse ( access . isValidToExecute ( null ) )
```

```
// Test method
testClone ( ) { org . apache . flink . api . common . accumulators . DoubleMinimum min = new org . apache . flink . api . common . accumulators .
DoubleMinimum ( ) ; double value = 3.14159265359 ; min . add ( value ) ; org . apache . flink . api . common . accumulators . DoubleMinimum clone =
min . clone ( ) ; "<AssertPlaceHolder>" ; }
// Focal method
getLocalValue ( ) { return null ; }

org . junit . Assert . assertEquals ( value , clone . getLocalValue ( ) , 0.0 )
```

Code summarization

```
public void update() { check Widget ( ) ; Utils . paintComponentImmediately ( handle ) ; update ( false ) ; }

forces all outstanding paint requests for the widget
```

```
public void setWordWrap(int row, int column, boolean wrap) { prepareCell ( row , column ) ; String wrapValue = wrap ? "" : "nowrap" ; DOM .
setStyleAttribute ( getElement ( row, column ) , "whiteSpace" , wrapValue ) ; }

sets whether the specified cell will allow word wrapping of its contents
```

Concerning the bug-fixing task, in the first example the model adds the `break` statement to each `case` of the `switch` block, thus allowing the program to break out of the `switch` block after one `case` block is executed. In the second example, instead, it changes the execution order of a statement as done by developers to fix the bug.

As per the mutants injection, the first example represents an *arithmetic operator deletion*, while the second is a *non void method call mutation*. While these transformations might look trivial, they are considered as correct since they reproduce real bugs that used to affect these methods. Thus, the model is basically choosing where to mutate and what to mutate in such a way to simulate real bugs (accomplishing one of the main goals of mutation testing).

Both examples of correct prediction we report involve the generation of an assert statement including an invocation to the focal method (*i.e.,* the main method tested by the test method). While the first is a rather "simple" `assertFalse` statement, the second required the guessing of the expected value (*i.e.,* `assertEquals`).

Finally, for the code summarization, the two reported examples showcase the ability of T5 to generate meaningful summaries equivalent to the ones manually written by developers.

3.3 Replication Package

Code and data used in the ICSE 2021 paper [1] and in the TSE paper [2] are publicly available:

- ICSE 2021: `https://github.com/antonio-mastropaolo/T5-learning-ICSE_2021`
- TSE 2022: `https://github.com/antonio-mastropaolo/TransferLearning4Code`

Chapter 4

Automating Logging Activities

This Chapter presents the status of our research in the context of automating logging activities. We start by summarizing our ICSE 2022 [29] paper presenting our first approach for log statements injection (Section 4.1). Then, we discuss our future research agenda on this topic (Section 4.2).

4.1 Using DL to Generate Complete Log Statements

Inspecting log messages is a popular practice that helps developers in several software maintenance activities such as testing [84, 85], debugging [86], diagnosis [87, 31], and monitoring [88, 89]. Developers insert log statements to expose and register information about the internal behavior of a software artifact in a human-comprehensible fashion [41]. The data generated is used for runtime and post-mortem analyses. For example, when debugging log statements can support root cause analysis [90, 91], while once the software is deployed logs can be used for performance monitoring [37] or anomaly detection [92, 93, 94].

Although technically possible, logging everything (*e.g.,* every exception) is inefficient and impracticable [36]. On the one hand, a coarse-grained logging risks hiding runtime failures, missing log messages useful for diagnoses [95], whereas a fine-grained logging risks increasing the overhead of log management and analysis [50]. To optimize the quantity and quality of data generated, developers insert log statements in strategic positions, specify appropriate log levels (*e.g.,* error, debug, info), and define compact but comprehensible text messages. Nonetheless, it remains a non-trivial task for developers to decide where, what, and at which level to log [50, 40].

To support software developers in these activities, we presented LANCE (Log stAtemeNt reCommEnder), the first approach able to synthesize and inject in the right position complete log statements, including a meaningful natural language message. LANCE is built on top of a T5 model [25] which has been trained on ∼7M Java methods. We collected Java methods from GitHub projects declaring an explicit dependency towards Apache Log4j [96], a well-known Java logging library. Such a design choice helped us constructing a more cohesive code-base, likely to include methods featuring log statements.

We start by pre-training the T5 model on a set of 6,832,859 Java methods. The pre-training has been performed through two pre-training objectives. The first is a classic "masked token" objective, in which we randomly mask 15% of the code tokens in the Java methods asking the model to guess them. This provides the model with general knowledge about the Java language.

The second pre-training objective provides as input to the model a Java method from which log statements originally present in it have been removed, with the T5 in charge of guessing where a log statement is needed by adding a special *<LOG_STMT>* token. This provides the model with additional knowledge about where logging is needed. Once pre-trained, the model is fine-tuned to generate complete log statements. In particular, we provide it with 61.5k Java methods from which we stripped a log statement originally written by developers and we ask LANCE to synthesize and inject the removed log statement. This means that LANCE must generate a complete log statement and inject it in the proper location, picking all the relevant log components: log level, code location and finally recommending a meaningful log message.

The trained model has then been run on a test set of previously unseen 7.1k Java methods, from which we removed also in this case one log statement, with LANCE in charge of re-injecting it. Our quantitative analysis showed that between the two pre-training strategies, a simple denoising task (*i.e.,* the model is asked to guess masked tokens in Java methods) allows T5 to achieve good performance. In particular, the best-performing model is able to (i) correctly predict the appropriate location of a log statement in 65.9% of cases; (ii) select a proper log level for the statement in 66.2% of cases; and (iii) generate a completely correct logging statement, including a meaningful natural language message in 15.2% of cases. The generation of the logging message basically acts as a sort of upper bound for the performance of LANCE, with 16.90% of generated logs having a correct message and an overall 15.20% of completely correct log statements (*i.e.,* level, position, and message are correct).

It is however worth noticing that we considered a prediction as correct only if the log statement was identical to the one originally written by the developers. However, when it comes to the log message which is written in natural language, it could happen that a predicted log statement includes a message different but semantically equivalent to the original one. For this reason, we manually inspected a sample of 300 "wrong" predictions in which the only "error" made by LANCE was the generated log message. The goal of the inspection was to understand whether the different log message was semantically equivalent to the original one. We found that a non-negligible set (~28%) of log messages classified as"wrong" actually contained the same information of the original log message, thus still being valuable recommendations.

Fig. 4.1 depicts examples of correct and "wrong" but meaningful recommendations generated by LANCE. In the first example ❶, LANCE injected a statement to log the state of the `msg` object. What it is interesting about this instance is that the model understood the need for invoking the method `ArrayConverter.bytesToHexString` in order to obtain a human comprehensible representation of the logged state. In the second instance ❷, LANCE inferred that if the `if` condition is not satisfied (*i.e.,* `value instanceof NSDictionary`), this indicates an unexpected value type for the passed parameter (`key`). In other words, the model mapped the `instanceof` operator to possible issues related to object types. The last correct prediction in Fig. 4.1 ❸ shows instead the ability of LANCE to compose log messages using the appropriate syntax needed to concatenate several parameters to string elements.

Moving to the "wrong" but semantically equivalent predictions (bottom part of Fig. 4.1), instance ❹ shows an interesting case in which the log message synthesized by LANCE (*i.e.,* "*Exception trying to delete subscription's configuration for subscription ID {}*") is even more detailed than the one written by developers (*i.e.,* "*Could not delete subscription for {}*). In ❺, instead, the opposite occurs (*i.e.,* the manually written message is more detailed) with the two messages, however, communicating similar information.

Figure 4.1. Examples of correct and wrong but meaningful predictions made by LANCE

Examples of correct predictions

1
```
public void writeSignatureHandshakeAlgorithm (CertificateRequestMessage msg){
        appendBytes(msg.getSignatureHashAlgorithm().getValues());
        LOGGER.debug("SignatureHashAlgorithm: " +
               ArrayConverter.bytesToHexString(msg.getSignatureHashAlgorithm().getValue()));
}
```

2
```
public NSDictionary objectForKey(final String key) {
    final NSObject value = dict.objectForKey(key);
    [...]

    if (value instanceof NSDictionary) {
        return (NSDictionary) value;
    }
    log.warn(String.format("Unexpected value type for serialized key %s", key));
    return null;
}
```

3
```
public ConnectionConsumer createConnectionConsumer(final Destination destination,
                              final ServerSessionPool pool, final int maxMessages) throws JMSException {
        if (ActiveMQRALogger.LOGGER.isTraceEnabled()) {
            ActiveMQRALogger.LOGGER.trace(
                      "createConnectionConsumer(" + destination + ", " + pool + ", " + maxMessages + ")");
        }
        throw new IllegalStateException(ISE);
}
```

Examples of "wrong" but semantically equivalent predictions (i.e., same information)

4
```
private synchronized CswSubscription deleteCswSubscription (String subscriptionId) throws CswException {

        [...]
        try {
            ServiceRegistration sr = registeredSubscriptions.remove(subscriptionId);
            [...]

        } catch (Exception e) {
            LOGGER.debug("Exception trying to delete subscription's configuration for subscription ID {}",
                          logSanitizedId, e);
                                                                          LANCE log statement

            LOGGER.debug("Could not delete subscription for {}", logSanitizedId, e);
                                                                          TARGET log statement
        }
}
```

5
```
public HandlerResult handle(ProcessState state, ProcessInstance process) {
        Secret secret = (Secret) state.getResource();
        String secretValue = secret.getValue();
        if (StringUtils.isNotBlank(secretValue)) {
            try {
                secretsService.delete(secret.getAccountId(), secret.getValue());
            } catch (IOException e) {
                log.error("Error deleting secret {}: {}", secret.getId(), e.getMessage());
                                                                          LANCE log statement
                log.error("Failed to delete secret from storage [{}]", secret.getId(), e);
                                                                          TARGET log statement
                [...]
            }
        }
        return null;
}
```

6
```
public static void sendApplicationInstanceActivatedEvent(String appId, String instanceId) {
        if (log.isInfoEnabled()) {
            log.info("Publishing application instance activated event: [application] " +
                     appId + " [instance] " + instanceId);
                                                                          LANCE log statement
            log.info("Publishing application instance active event: [application]" +
                     appId + "[instance]" + instanceId) ;
                                                                          TARGET log statement
        }
        [...]
}
```

Finally, the last example ❻ shows two messages only differing for one word (*activated vs active*). This example is representative of many instances we found in which differences were even smaller.

For example, we observed cases in which the only difference was the usage of letter case. Indeed, in our quantitative analysis we decided to be conservative, considering a prediction as correct only if it matched the reference one even in terms of letter case. This was done to avoid considering as correct predictions making a wrong usage of camelCase.

4.2 Future Directions

Despite the encouraging results, we acknowledge that LANCE is just a first attempt at automatically generating complete log statements, and additional improvements are needed. We are now working towards addressing two practical limitations of our model.

First, while LANCE can generate complete log statements, it cannot decide whether a log statement is needed in a given method. This is due to the specific training that we performed. Indeed, we only showed to the T5 model instances (*i.e.,* Java methods) requiring the injection of exactly one log statement. This implies that, when provided with a Java method as input, LANCE always assumes that a single log statement must be injected. However, in practice, this is clearly not the case. Such a training strategy is also the reason for the second limitation of LANCE: It can only inject one single log statement in a given method. However, a method may require multiple log statements.

Thus, we plan to train LANCE with the goal of being able to (i) understand whether log statements are actually needed or not in a given method; and (ii) in case of positive answer, to generate and inject as many log statements as needed.

Concerning the first point, we are considering training a DL-based binary classifier capable of discriminating whether logs are needed in a method provided as input. In doing so, the output of the classifier would be a label (*e.g., Need/No Need*) indicating the need for logs. As per the multi-log injection support, our idea is to perform a training similar to the one we have adopted for LANCE, with the difference being in the construction of dataset needed for training, evaluating and testing the model. In detail, we can proceed stripping more than one log statements from a Java method asking the model to generate all the log statements that have been removed from the method. When the model reaches convergence –ideally– it would be capable of injecting an arbitrary number of log statements ($N \geq 1$) in a Java method provided as input. By integrating the binary classifier and the multi-log injection model, we could be able to inject an arbitrary number of log statements only when needed.

Finally, we plan to integrate the newer version of LANCE in an IDE plugin (likely a vscode plugin), to run controlled experiments with developers about its usefulness.

4.3 Replication Package

The code and data used in the ICSE 2022 paper [29] are publicly available at: `https://github.com/antonio-mastropaolo/LANCE`.

Chapter 5

Code Summarization

Section 5.1 summarizes the status of our research in the context of code summarization, presenting our ICSME 2021 paper [28]. Directions for future work are discussed in Section 5.2.

5.1 An Empirical Study on Code Comment Completion

Code comments play a prominent role in program comprehension activities. However, source code is not always documented and code and comments not always co-evolve. To deal with these issues, researchers have proposed techniques to automatically generate comments documenting a given code at hand. Despite the achieved advances using DL techniques [14, 71, 13, 58, 70, 68, 1], the empirical evaluations of these approaches show that they are still far from a performance level that would make them valuable for developers. For instance, Haque *et al.* [14] were capable of documenting Java methods as a real developer would do only in ∼3% of cases. For such a reason, we decided to start tackling a simpler and related problem: Code comment completion. Instead of generating a comment for a given code from scratch, we investigate the extent to which a simple n-gram model and the T5 architecture can perform in autocompleting a code comment the developer is typing. To this end, we instantiate the two models to the problem of code comment completion in Java. We tackle the problem at method-level granularity, meaning that we expect the model to learn how to autocomplete a code comment used to document a method or part of it.

The context of our study is represented by a dataset composed by 497,328 Java methods with their related comments. We pre-train T5 using a self-supervised task similar to the one used by Raffel *et al.* [25], consisting of masking tokens in natural language sentences and asking the model to guess the masked tokens. Since we want the model to learn how to autocomplete comments given a certain coding context (*i.e.,* a specific Java method), we randomly mask 15% of tokens appearing in the comment-related part of each instance. Tokens representing the method code were not masked. Once pre-trained, we fine-tune the T5 model in a multi-task setting, in which the two tasks are represented in our case by the automatic completion of (i) Javadoc comments and (ii) inner comments.

The achieved results show that T5 outperforms the n-gram model, achieving superior performance in all the comment completion scenario we tested. This is especially true when the two models are required to auto-complete a limited number of tokens ($k < 7$) following the ones already written by the developer in the code comment. For example, when only the subsequent word must be predicted (*i.e.,* $k = 1$), T5 can achieve >50% of correct predictions when completing a Javadoc and >25% for inner comments while the n-gram model, in both scenarios, achieves less than 16% of correct predictions.

For a complete quantitative and qualitative discussion of the achieved results, we point the reader to the corresponding paper [28].

In summary, the results of our study showed that code comment completion is definitively a simpler problem as compared to whole comment generation. Indeed, when working at method-level granularity, our ICSE'21 paper showed that T5 can generate correct summaries for ∼11% of provided methods, while in the context of comment completions the model can push its performance up to 16% even with 10 tokens to predict following the ones already typed by the developer. Still, 16% is quite far from what could be defined as an acceptable result to deploy these tools to developers. Our assumption is that summarizing a whole method is too complex given the datasets and models currently available. For this reason, we decided to shift our future research efforts to snippet-level summarization (*i.e.,* automatically documenting a few lines of code) as described in Section 5.2.

5.2 Future Directions

Working on DL-based snippet-level summarization brings as first challenge the building of a training dataset: While it is easy to create datasets composed of pairs <method, javadoc> to support method-level summarization, this is not the case for code snippets. Indeed, it is not trivial to identify the code lines documented by an inner comment. While simple heuristics have been defined in the literature (*e.g.,* an inner comment documents all statements following it until a black line is found), these heuristics fail a substantial amount of times based on our analyses. Thus, our plan is to (i) train a DL model to link inner comments and code snippets, and (ii) use the trained model to build a large-scale dataset of <*snippet*, *description*> pairs that can then be used to train another model in charge of snippet-level summarization.

We started by manually building a dataset of <*snippet*, *description*> pairs, in which we classified the code comment (*description*) as being or not a code summary and linked it to the documented Java statements. As a first step to build our dataset we needed to collect the set of code comments D_1, D_2, ..., D_n to manually analyze. These comments have been collected from 100 GitHub Java projects having at least 500 commits, 25 contributors, 10 stars, and not being forks. These filters aim at discarding personal/toy projects and reducing the chance of mining duplicated code. We parsed their code to identify comments within each method to manually analyze. We ignored Javadoc comments since they document entire methods rather than code snippets: We only considered single-line (starting with "`//`") and multi-line (starting with "`/*`") comments as subject of our manual analysis. The manual analysis has been performed by six researchers (from now on, evaluators) through a web app we developed. The web app assigned each Java file to two evaluators who independently labeled the comments in it. The goal of the labeling was to firstly assign the comment D to one or more categories CCs. The starting set of categories to use was taken from the work by Pascarella *et al.* [83] and included: *summary*, *rationale*, *deprecation*, *usage*, *exception*, *TODO*, *incomplete*, *commented code*, *formatter*, and *pointer*. We point the reader to [83] for a complete description of these categories. Once defined the category for a given comment under analysis, the next step was the linking of the comment to the documented code DC. The linking has been performed at line-level granularity. This means, for example, that for a comment D the evaluator could indicate lines 11, 12, and 17 as documented. Note that gaps are possible in DC (*i.e.,* the documented code could be composed by non-contiguous lines).

Then, we started resolving conflicts arisen from the manual analysis. Two types of conflicts are possible for each manually defined triplet <D, $\{CC\}$, DC>:

The two evaluators could have (i) selected a different set $\{CC\}$ when classifying the comment; and (ii) identified different sets of lines (DC) documented by the comment. Out of the 11,563 manually labeled comments, 2,444 (21%) resulted in a conflict: 2,002 were due to different comment categories selected by the evaluators; 83 to differences in the selected DC; 359 concerned both the categories and the DC. Conflicts were solved by a third evaluator not involved in the labeling of the conflicting instance. Overall, we spent 1,419 man-hours on the labeling and conflict resolution, manually annotating 11,563 comments (with two evaluators for each of them) coming from 1,508 Java files and 85 software projects.

Table 5.1. Dataset output of manual labeling

Category	**#Instances**	**Documented Statements**		
		mean	**median**	**sd**
Summary	5,427	2.36	1.0	2.78
Formatting	4,635	1.67	1.0	2.64
Rationale	1,486	2.22	1.0	3.18
Commented Code	569	0.00	0.0	0.00
TODO	480	0.33	0.0	0.90
Orphan	66	0.00	0.0	0.00
Pointer	59	1.47	1.0	3.90
Code Example	32	1.50	1.0	1.83
Incomplete	16	0.43	0.0	0.63
Deprecation	12	3.16	2.0	2.92
Overall	**11,563**	**1.31**	**0.7**	**1.88**

Table 5.1 summarizes the dataset obtained as output of our analysis. Besides reporting the categories to which the comments in our dataset belong, Table 5.1 also shows descriptive statistics related to the number of statements documented by comments belonging to different categories. As expected, *orphan* and *commented code* comments are not linked to any code statement. More than half of *TODO* comments is also not linked to any statement, since in many cases todos are related to *e.g.,* feature that must be implemented. The most frequent category is, as expected, the *summary* one (5,427 instances) grouping comments summarizing one or more code statements (on average, 2.36 statements). Another popular category is "*formatting*", with 4,635 instances. While one could expect no code linked to formatting comments, this is actually not the case since we used such a category also for comments nor adding new information to the documented code but just acting as a logical split of the code (*e.g.,* a comment `//get messages` put on top of a method call `getMessages()`). Finally, comments explaining the *rationale* for implementation choices account for 1,486 instances. While we focus on the generation of code summaries, these instances contains information that is hard to automatically synthesize and could represent a seed for future research.

Interestingly, 3,250 of the comments in our dataset (∼28%) include "gaps" in the linked code. This means, for example, that a comment documents lines 11, 12, and 17 (but not lines 13-16). Thus, approaches to automatically link comment and code must take such a scenario into account. Also, in ∼70% of cases the documented code is not followed by an empty line: A simple heuristic that just links a given comment to all following lines until an empty line is found would succeed in ∼30% of cases. These observations highlight the need for a more complex comment-to-code linking approach. Our future work will focus on: (i) using our dataset to train a DL model able to link inner comments to the code snippet they document; and (ii) run such a model on thousands of projects to build a large-scale dataset of <*snippet*, *description*> pairs that can be used to train a model for snippet-level code summarization.

Chapter 6

Conclusions and Future Plans

In this thesis proposal, I described our research in the context of automating code-related tasks characterized by the presence of bi-modal data via pre-trained language models of code. We first investigated the performance of T5 in supporting code-related tasks [1, 2]. Given the promising results achieved, we instantiated T5 to the task of: (i) injection and generation of complete log statements, and (ii) code summarization. As for the former, we presented *LANCE* [29], the first approach capable of helping developers in all log-related activities (*i.e.,* writing the log message, placing the log statement, and choosing the appropriate log level). We aim at improving LANCE by integrating in it the ability to decide whether logs are actually needed in a given method and, if that is the case, to inject into it all needed log statements.

As per the code summarization task, we found out that even when using pre-trained language models of code, the sought-after performance needed to consider these tools useful for developers is challenging to achieve if the model is asked to document a complete method [1]. For such a reason, we started working towards investigating how pre-trained models of code perform when generating summaries for smaller code components (*i.e.,* code snippets).

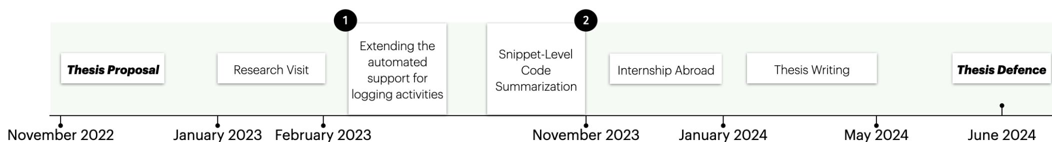

Figure 6.1. PhD. Plan Timeline

Fig. 6.1 shows the plan for the remaining part of my PhD. From *January 2023 to February 2023* I will be visiting professor Massimiliano di Penta at the *Università degli Studi del Sannio*. From *March 2023 to November 2023*, I plan to further work on extending the automated support for logging, while developing a code-snippet summarizer using the manually-labeled dataset we built. From *November 2023 to January 2024* I also plan an internship period abroad. Finally, I will focus on the thesis writing starting from *February 2024*.

Bibliography

[1] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 336–347, IEEE, 2021.

[2] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, 2022.

[3] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE software*, vol. 27, no. 4, pp. 80–86, 2009.

[4] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 230–232, IEEE, 2013.

[5] Y. Malheiros, A. Moraes, C. Trindade, and S. Meira, "A source code recommender system to support newcomers," in *2012 ieee 36th annual computer software and applications conference*, pp. 19–24, IEEE, 2012.

[6] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Arena: an approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, 2016.

[7] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *2013 IEEE Sixth international conference on software testing, verification and validation*, pp. 282–291, IEEE, 2013.

[8] "Github statistics." https://en.wikipedia.org/wiki/GitHub, n.d.

[9] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[10] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1943–1959, sep 2021.

[11] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: learning to repair compilation errors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 925–936, 2019.

[12] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *arXiv preprint arXiv:1812.07170*, 2018.

[13] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, pp. 184–195, 2020.

[14] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 300–310, 2020.

[15] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 184–195, IEEE, 2021.

[16] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

[17] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, 2021.

[18] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2727–2735, 2019.

[19] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 473–485, 2020.

[20] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," *arXiv preprint arXiv:2202.06689*, 2022.

[21] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[22] R. Robbes and A. Janes, "Leveraging small software engineering data sets with pre-trained neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 29–32, IEEE, 2019.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, pp. 5998–6008, Curran Associates, Inc., 2017.

[24] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[25] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2019.

[26] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 301–312, IEEE, 2019.

[27] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1398–1409, 2020.

[28] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 159–170, IEEE, 2021.

[29] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 2279–2290, ACM, 2022.

[30] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

[31] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–28, 2012.

[32] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 2012–2031, 2019.

[33] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "Smartlog: Place error log statement by deep understanding of log intention," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 61–71, IEEE, 2018.

[34] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, 2018.

[35] Z. Li, "Towards providing automated supports to developers on writing logging statements," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pp. 198–201, 2020.

[36] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 415–425, IEEE, 2015.

[37] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting logging locations for web-based systems' performance monitoring," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 127–138, 2018.

[38] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 361–372, 2020.

[39] J. Cândido, J. Haesen, M. Aniche, and A. van Deursen, "An exploratory study of log placement recommendation in an enterprise system," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 143–154, IEEE, 2021.

[40] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th international conference on software engineering (ICSE)*, pp. 102–112, IEEE, 2012.

[41] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[42] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue, "Padla: a dynamic log level adapter using online phase detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 135–138, IEEE, 2019.

[43] Z. Li, H. Li, T.-H. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1461–1472, IEEE, 2021.

[44] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 24–33, 2014.

[45] B. Chen *et al.*, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, 2017.

[46] Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on f-droid," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3394–3434, 2019.

[47] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, and T. Xie, "An exploratory study of logging configuration practice in java," in *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 459–469, IEEE, 2019.

[48] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, 2018.

[49] R. Zhou, M. Hamdaqa, H. Cai, and A. Hamou-Lhadj, "Mobilogleak: A preliminary study on data leakage caused by poor logging practices," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 577–581, IEEE, 2020.

[50] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, 2020.

[51] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*, pp. 35–44, IEEE, 2010.

[52] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *2011 IEEE 19th International Conference on Program Comprehension*, pp. 71–80, IEEE, 2011.

[53] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 23–32, IEEE, 2013.

[54] P. Rodeghero, S. Jiang, A. Armaly, and C. McMillan, "Detecting user story information in developer-client conversations to generate extractive summaries," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 49–59, IEEE, 2017.

[55] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th international conference on Software engineering*, pp. 390–401, 2014.

[56] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 101–110, IEEE, 2011.

[57] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2015.

[58] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20010, IEEE, 2018.

[59] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

[60] S. Banerjee and A. Lavie, "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*, (Ann Arbor, Michigan), pp. 65–72, Association for Computational Linguistics, June 2005.

[61] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, “Towards automatically generating summary comments for java methods,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52, 2010.

[62] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, “Improving topic model source code summarization,” in *Proceedings of the 22nd international conference on program comprehension*, pp. 291–294, 2014.

[63] N. Dragan, M. L. Collard, and J. I. Maletic, “Reverse engineering method stereotypes,” in *2006 22nd IEEE International Conference on Software Maintenance*, pp. 24–34, IEEE, 2006.

[64] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, “Autofolding for source code summarization,” *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1095–1109, 2017.

[65] N. Dragan, M. L. Collard, and J. I. Maletic, “Automatic identification of class stereotypes,” in *2010 IEEE International Conference on Software Maintenance*, pp. 1–10, IEEE, 2010.

[66] B. Li, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, “Aiding comprehension of unit test cases and test suites with stereotype-based tagging,” in *Proceedings of the 26th Conference on Program Comprehension*, pp. 52–63, 2018.

[67] Y. Huang, S. Huang, H. Chen, X. Chen, Z. Zheng, X. Luo, N. Jia, X. Hu, and X. Zhou, “Towards automatically generating block comments for code snippets,” *Information and Software Technology*, vol. 127, p. 106373, 2020.

[68] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “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)*, pp. 2073–2083, 2016.

[69] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv preprint arXiv:1409.0473*, 2014.

[70] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” *arXiv preprint arXiv:2005.00653*, 2020.

[71] H. Wu, H. Zhao, and M. Zhang, “Code summarization with structure-induced transformer,” *arXiv preprint arXiv:2012.14710*, 2020.

[72] A. Eriguchi, K. Hashimoto, and Y. Tsuruoka, “Tree-to-sequence attentional neural machine translation,” *arXiv preprint arXiv:1603.06075*, 2016.

[73] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, “Improving automatic source code summarization via deep reinforcement learning,” in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pp. 397–407, 2018.

[74] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” *Advances in neural information processing systems*, vol. 27, 2014.

[75] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.," in *IJCAI*, pp. 3034–3040, 2017.

[76] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 81–90, IEEE, 2015.

[77] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 380–389, IEEE, 2015.

[78] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 562–567, IEEE, 2013.

[79] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza, "Automated documentation of android apps," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 204–220, 2019.

[80] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatically generating natural language descriptions for object-related statement sequences," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 205–216, IEEE, 2017.

[81] W. Zheng, H. Zhou, M. Li, and J. Wu, "Codeattention: translating source code to comments by exploiting the code constructs," *Frontiers of Computer Science*, vol. 13, no. 3, pp. 565–578, 2019.

[82] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.

[83] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 227–237, IEEE, 2017.

[84] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 305–316, 2018.

[85] J. Chen, W. Shang, A. E. Hassan, Y. Wang, and J. Lin, "An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 669–681, IEEE, 2019.

[86] M. Satyanarayanan, D. C. Steere, M. Kudo, and H. Mashburn, "Transparent logging as a technique for debugging complex distributed systems," in *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pp. 1–3, 1992.

[87] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 683–694, 2019.

[88] W. Hasselbring and A. van Hoorn, "Kieker: A monitoring framework for software engineering research," *Software Impacts*, vol. 5, p. 100019, 2020.

[89] J. Harty, H. Zhang, L. Wei, L. Pascarella, M. Aniche, and W. Shang, "Logging practices with mobile analytics: An empirical study on firebase," *Proceedings of the 2021 8th International Conference on Mobile Software Engineering and Systems (MOBILESoft-2021)*, 2021.

[90] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for spark," in *2017 IEEE International Conference on Web Services (ICWS)*, pp. 389–396, IEEE, 2017.

[91] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne, "Crude: Combining resource usage data and error logs for accurate error detection in large-scale distributed systems," in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pp. 51–60, IEEE, 2016.

[92] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.," in *IJCAI*, vol. 19, pp. 4739–4745, 2019.

[93] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 807–817, 2019.

[94] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1285–1298, 2017.

[95] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pp. 143–154, 2010.

[96] *Apache Log4j*. https://logging.apache.org/log4j/2.x/.