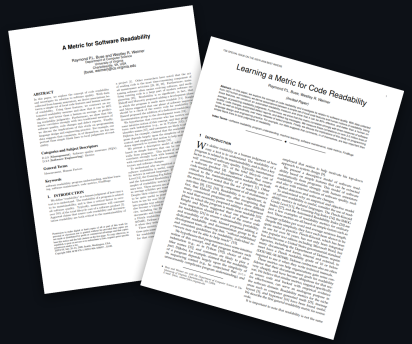
USING DEEP LEARNING TO AUTOMATICALLY IMPROVE CODE READABILITY

Antonio Vitale Valentina Piantadosi **Simone Scalabrino** Rocco Oliveto

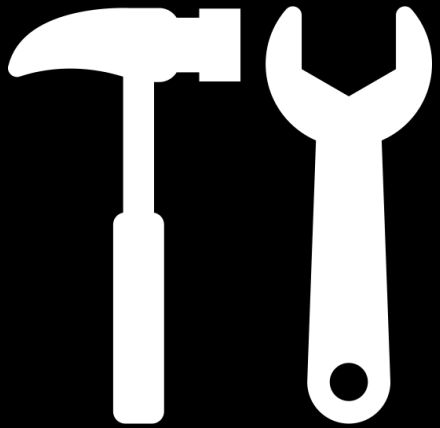
Code Readability

Assessing code readability



What if code is **unreadable**?

Other tools

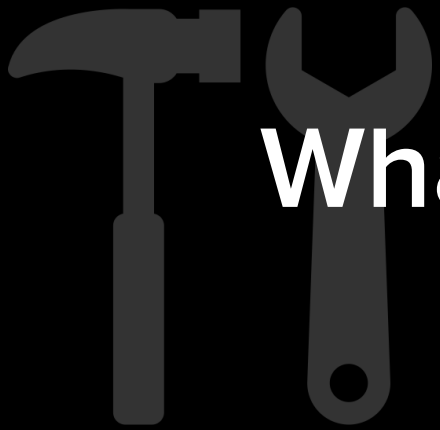


Other tools

Formatters

Style improvement

Rename refactoring tools



What ones should you use?

Formatters

Style improvement

Rename refactoring tools

Other tools

Other tools

Manual improvement

Ideally...

```
public static int cw(String f) {
    int wc = 0;
    for (String w:f.split(" ")) {wc++;}
    return wc;
}
```

Ideally...

```java
public static int cw(String f) {
    int wc = 0;
    for (String w:f.split(" ")) {wc++;}
    return wc;
}
```

```java
public static int countWords(String fileContent) {
    int wordCount = 0;
    for (String word : fileContent.split(" ")) {
        wordCount++;
    }

    return wordCount;
}
```

Our goal is to introduce an approach for improving code readability as a whole

Bug-fixing

Code summarization

Generation of assert statements

Mutants generation

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‡, Rocco Oliveto†, Gabriele Bavota*
*SEART @ Software Institute, Università della Svizzera italiana (USI), Switzerland
†University of Molise, Italy
‡SEMERU @ Computer Science Department, William and Mary, USA

***Abstract*—Deep learning (DL) techniques are gaining more and more attention in the software engineering community. They have been used to support several code-related tasks, such as automatic bug fixing and code comments generation. Recent studies in the Natural Language Processing (NLP) field have shown that the Text-To-Text Transfer Transformer (T5) architecture can achieve state-of-the-art performance for a variety of NLP tasks. The basic idea behind T5 is to first pre-train a model on a large and generic dataset using a self-supervised task (*e.g.*, filling masked words in sentences). Once the model is pre-trained, it is fine-tuned on smaller and specialized datasets, each one related to a specific task (*e.g.*, language translation, sentence classification). In this paper, we empirically investigate how the T5 model performs when pre-trained and fine-tuned to support code-related tasks. We pre-train a T5 model on a dataset composed of natural language English text and source code. Then, we fine-tune such a model by reusing datasets used in four previous works that used DL techniques to: (i) fix bugs, (ii) inject code mutants, (iii) generate assert statements, and (iv) generate code comments. We compared the performance of this single model with the results reported in the four original papers proposing DL-based solutions for those four tasks. We show that our T5 model, exploiting additional data for the self-supervised pre-training phase, can achieve performance improvements over the four baselines.**

***Index Terms*—Empirical software engineering, Deep Learning**

I. Introduction

Deep Learning (DL) has been used to support a vast variety of code-related tasks. Some examples include automatic bug fixing [1]–[4], learning generic code changes [5], code migration [6], [7], code summarization [8]–[11], pseudo-code generation [12], code deobfuscation [13], [14], injection of code mutants [15], automatic generation of assert statements [16], and code completion [17]–[21]. These works customize DL models proposed in the Natural Language Processing (NLP) field to support the previously listed tasks. For instance, Tufano *et al.* [1] used an RNN Encoder-Decoder architecture, commonly adopted in Neural Machine Translation (NMT) [22]–[24], to learn how to automatically fix bugs in Java methods. The model learned bug-fixing patterns by being trained on pairs of buggy and fixed methods mined from software repositories. This work, as the vast majority of the ones previously mentioned (*e.g.*, [5], [6], [11], [15], [16]), share one common characteristic: *They shape the problem at hand as a text-to-text transformation, in which the input and the output of the model are text strings.*

For example, in the work by Watson *et al.* [16] the input is a string representing a test method without an assert statement, and the output is an appropriate assert statement for the given test. In the approach by Haque *et al.* [11], the input is composed of strings representing a subroutine to document, while the output is a natural language summary documenting the subroutine.

Recent years have seen the raise of transfer learning in the field of natural language processing. 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 (Text-To-Text Transfer Transformer) 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.

The goal of this work is to empirically investigate the potential of a T5 model when pre-trained and fine-tuned to support many of the previously listed code-related tasks also characterized by text-to-text transformations. We started by pre-training a T5 model using a large dataset consisting of 499,618 English sentences and 1,569,889 source code components (*i.e.*, 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.* [1], 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.

Injection of code mutants. This dataset is also by Tufano *et al.* [15], 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, while the output is its buggy version). The model must learn how to inject bugs (mutants) in code instead of fixing bugs.

Generation of assert statements in test methods. We use the dataset by Watson *et al.* [16], 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.

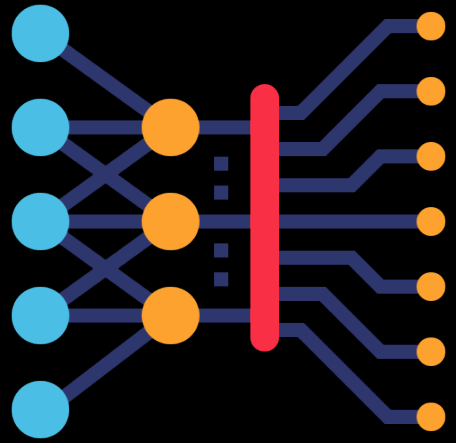
Mastropaolo *et al.* 2021

Bug-fixing

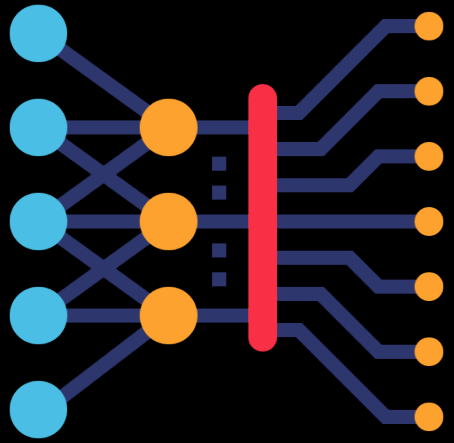
Code summarization

Generation of assert statements

Mutants generation



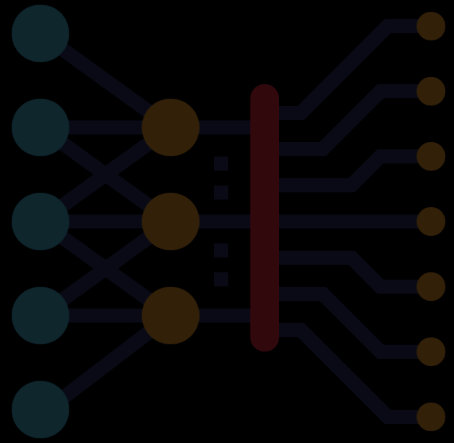
Using a LLM-based approach to improve readability



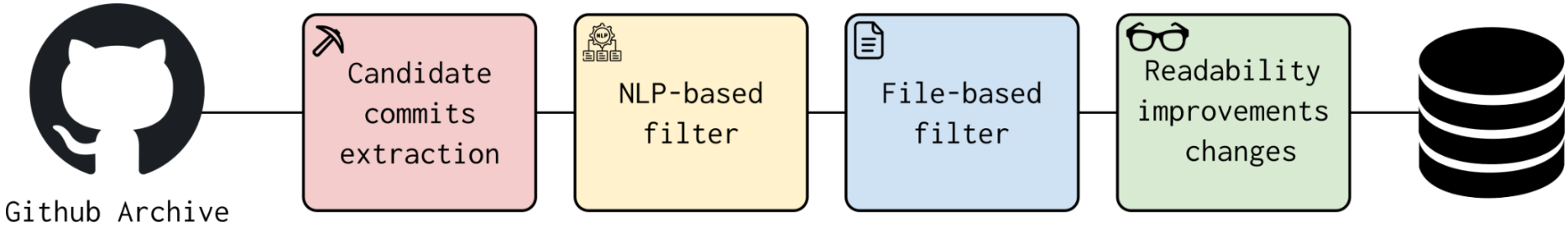
Dataset of
readability-improving chages

Using a LLM-based approach to
improve readability

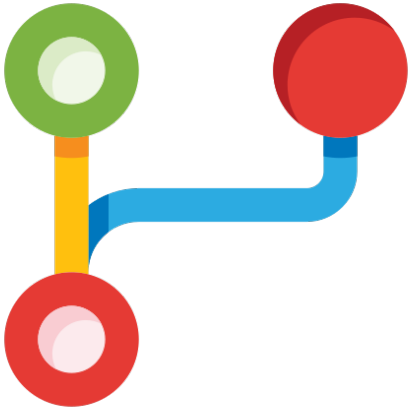
Dataset of
readability-improving chages



Using a LLM-based approach to
improve readability

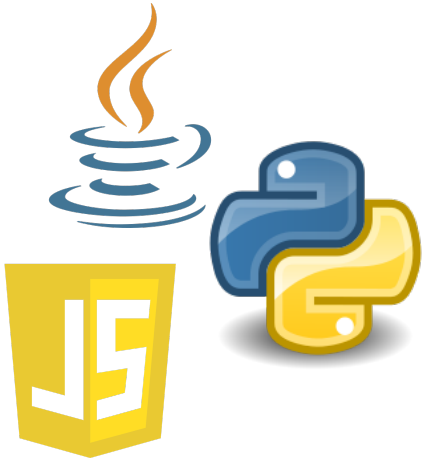


Between 2015 to 2022



122k commits

156k files



10 languages

How precise is our approach in identifying readability-improving commits?

RQ1

RQ1

500 commits

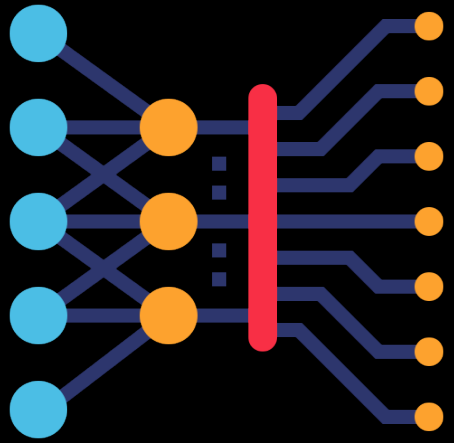
RQ1

82% - 91% accuracy

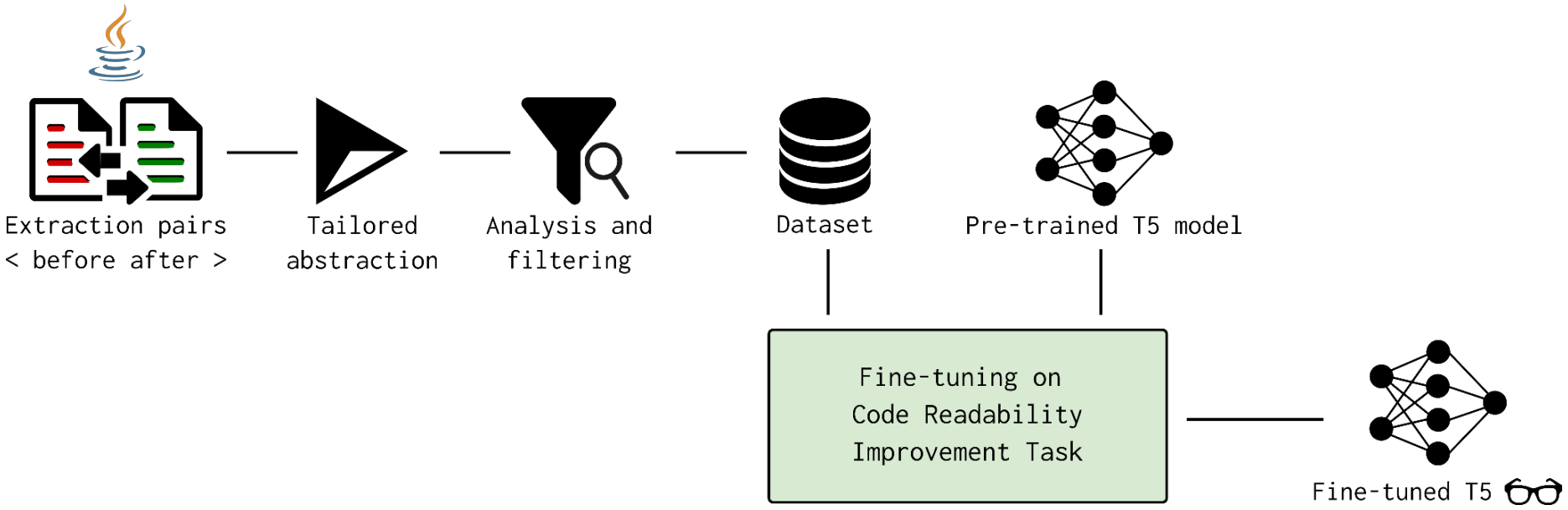
95% confidence level

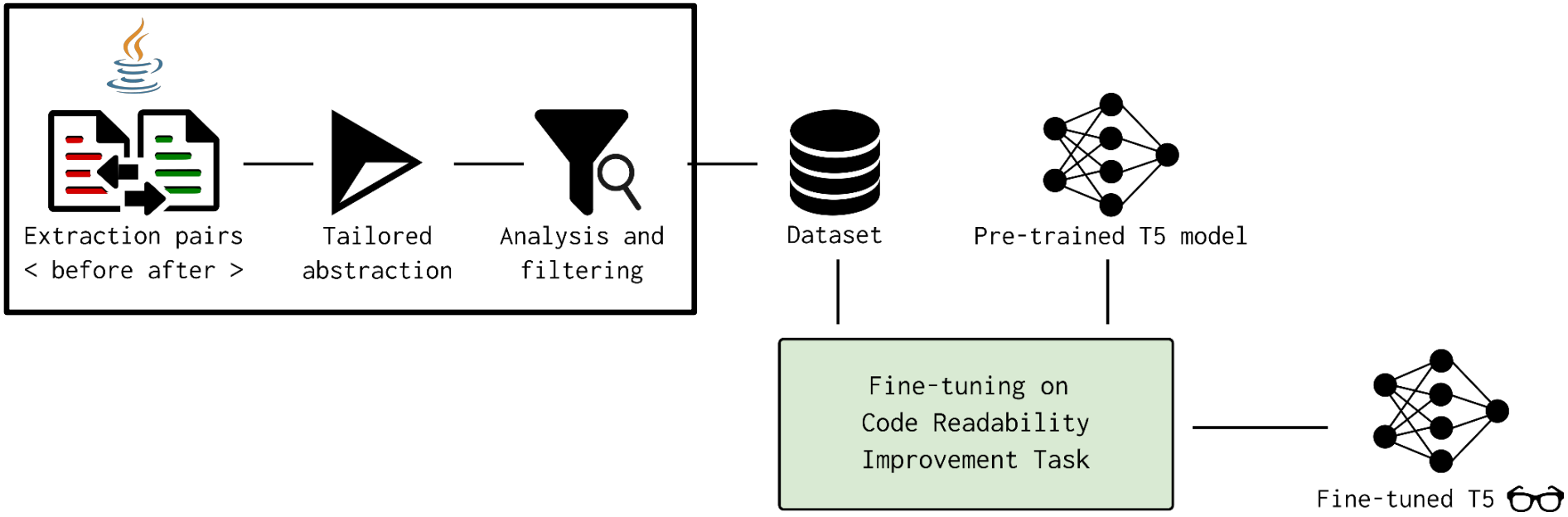
500 commits

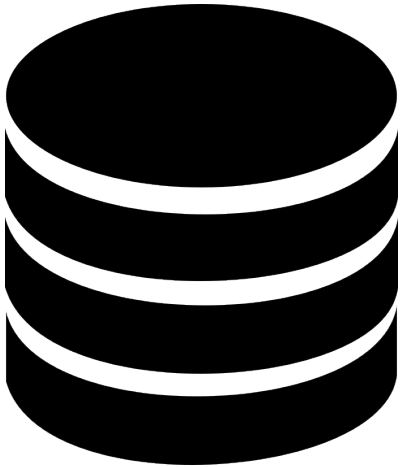
Dataset of
readability-improving chages



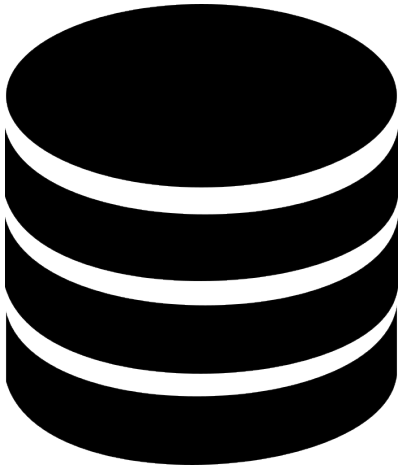
Using a LLM-based approach to
improve readability


Extraction pairs
< before after >
Tailored
abstraction
Analysis and
filtering
Dataset
Pre-trained T5 model
Fine-tuning on
Code Readability
Improvement Task
Fine-tuned T5


Extraction pairs
< before after >
Tailored
abstraction
Analysis and
filtering
Dataset
Pre-trained T5 model
Fine-tuning on
Code Readability
Improvement Task
Fine-tuned T5

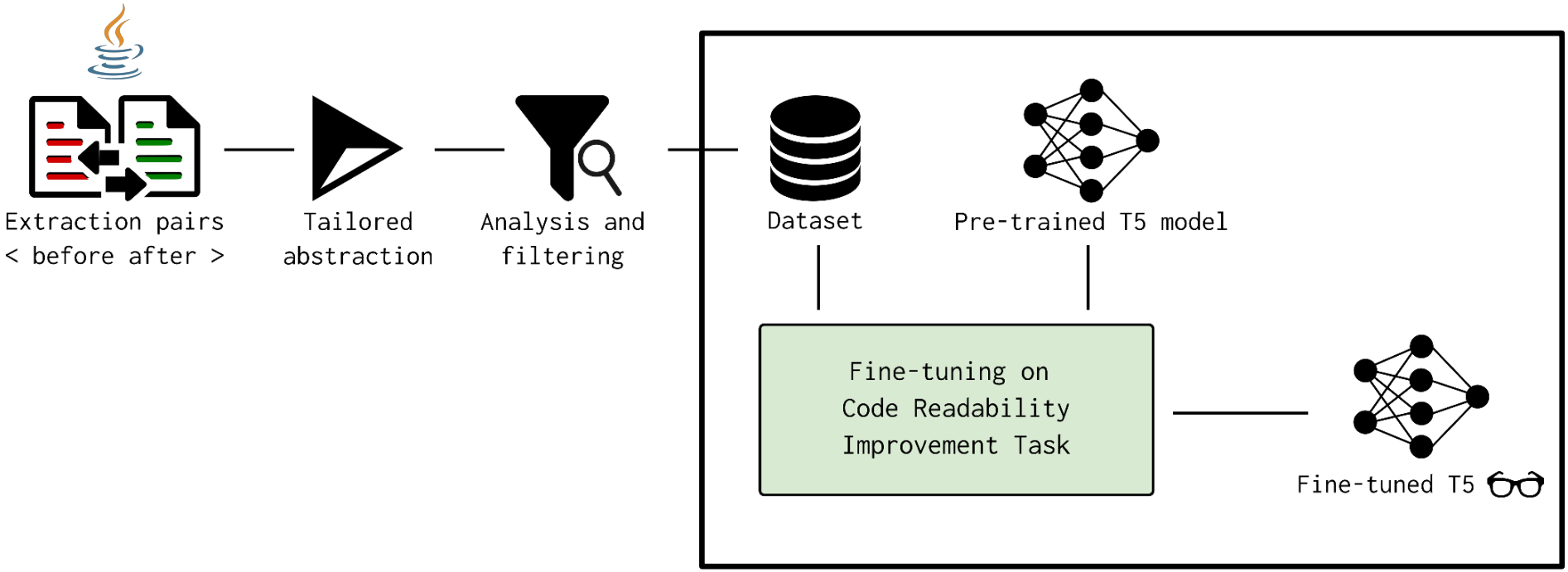


31,183 pairs

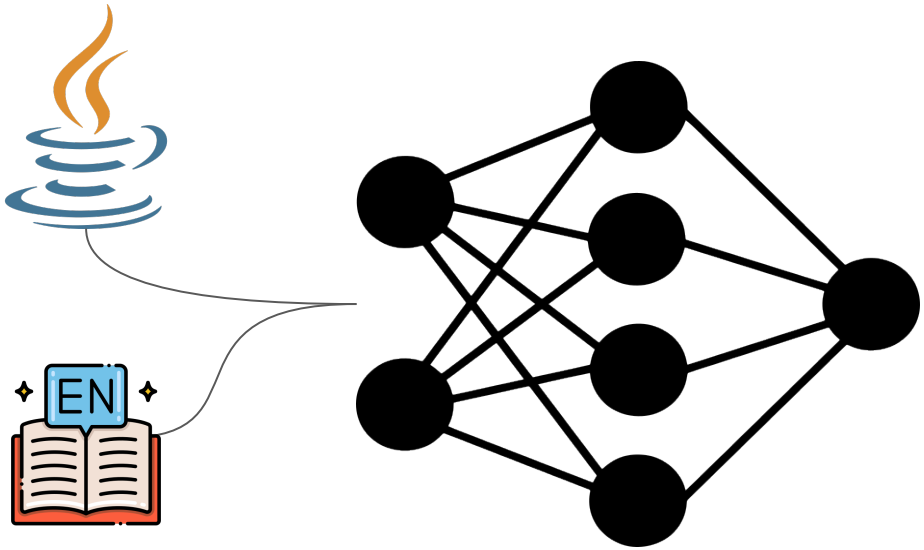


31,183 pairs

80% Training
10% Validation
10% Test

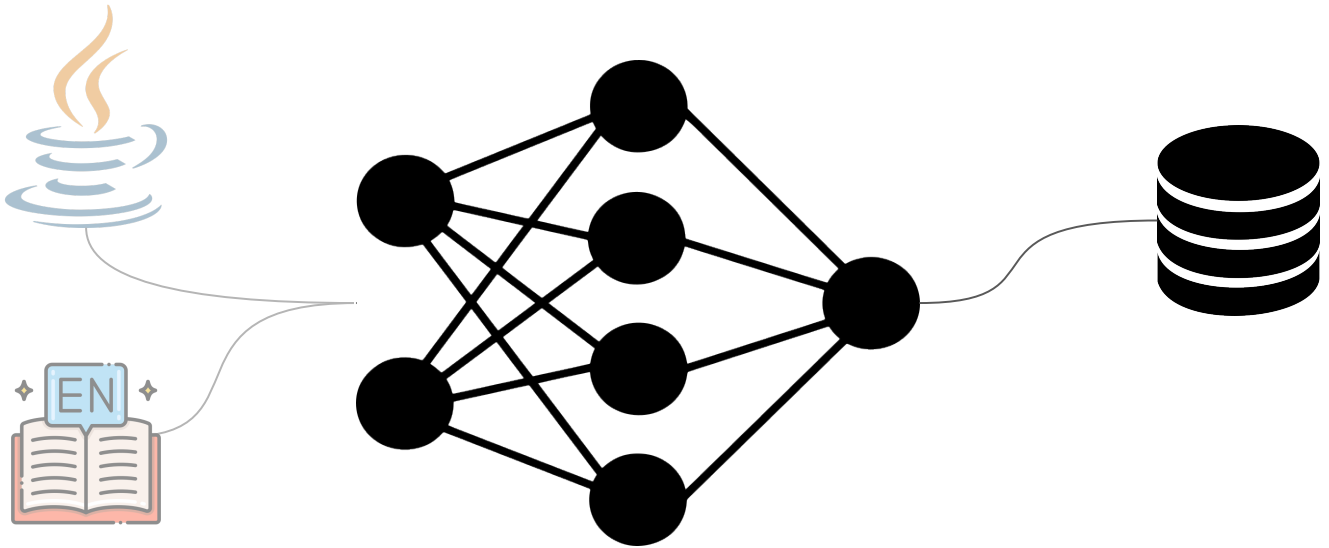

Extraction pairs
< before after >
Tailored
abstraction
Analysis and
filtering
Dataset
Pre-trained T5 model
Fine-tuning on
Code Readability
Improvement Task
Fine-tuned T5

T5 model



Pre-training

T5 model

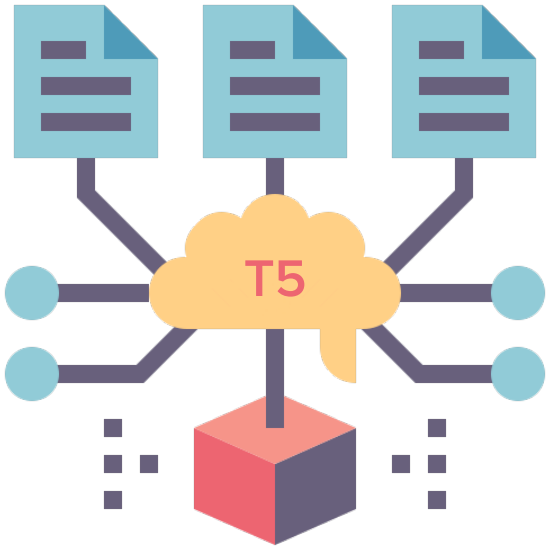


Fine tuning

How does our model perform compared to models for other coding tasks?

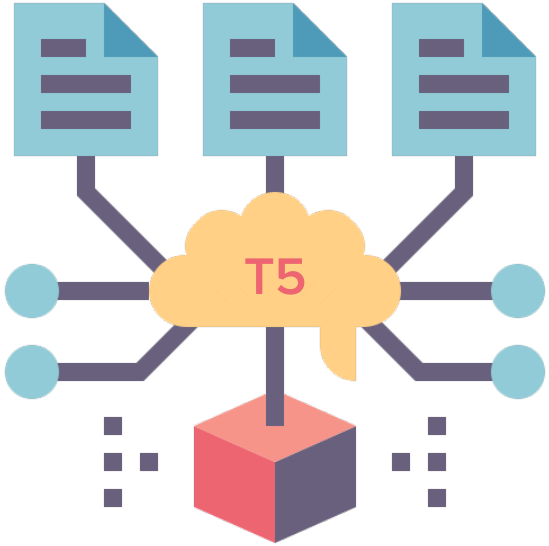
RQ2

RQ2

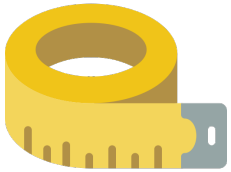


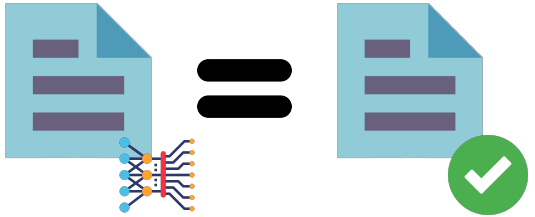
Beam Search

RQ2

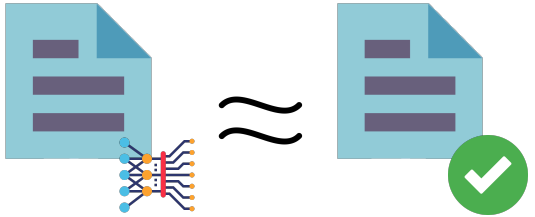


Beam Search





Accuracy@K



BLEU-A

RQ2

Accuracy@1

21%

10%	3%	34%	47%	28%
Bug Fixing (small)	Bug Fixing (medium)	Assertion Generation (raw)	Assertion Generation (abstracted)	Mutant Generation

RQ2

BLEU-A

0.77

0.78

Mutant
Generation

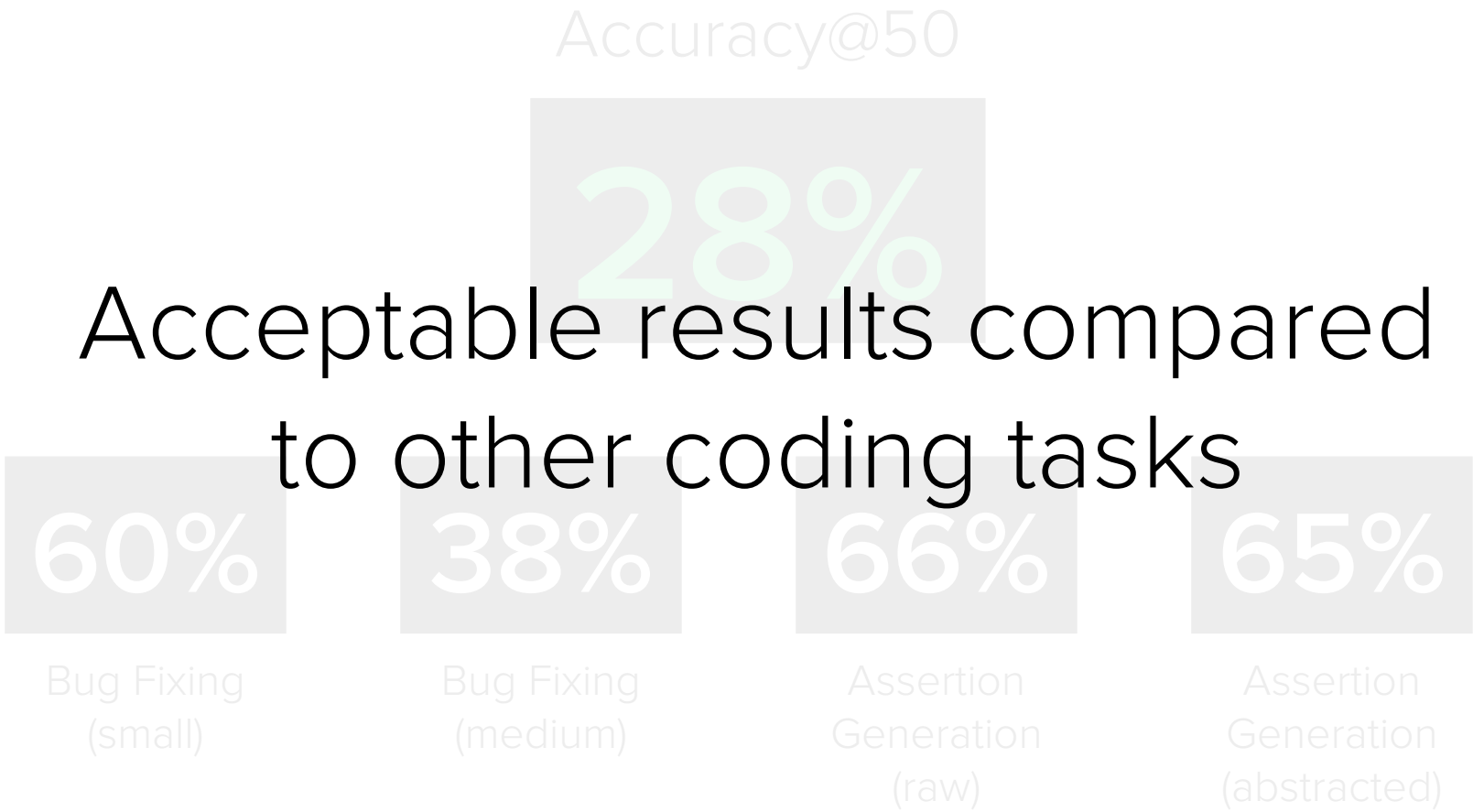
RQ2

Accuracy@50

28%

60%	38%	66%	65%
Bug Fixing (small)	Bug Fixing (medium)	Assertion Generation (raw)	Assertion Generation (abstracted)

RQ2



Acceptable results compared to other coding tasks

To what extent does our model try to improve code readability?

RQ3

RQ3

500 instances

127 perfect predictions

373 non-perfect predictions

"Does the predicted modification try to improve code readability?"

RQ3

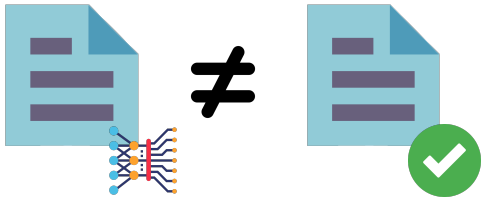
Tries to improve readability

34%

RQ3

Tries to improve readability

34%



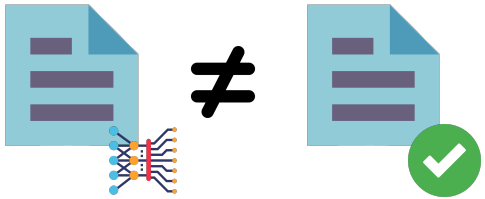
Non-perfect predictions

19%

RQ3

Tries to improve readability

34%



Non-perfect predictions

19%

RQ3

Non-perfect predictions

47%

the model does not change the code

RQ3

Overall

69%

the model does not change the behavior

RQ3

Overall

69%

The model is reliable enough

the model does not change the behavior

To what extent does the proposed model improve code readability?

RQ4

RQ4

“How does the change impact code readability?”

RQ4

79%

cases in which the readability increases

RQ4

7%

cases in which the readability decreases

RQ4

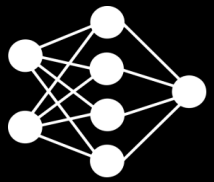
The model actually improves code readability

7%

cases in which the readability decreases

Methodology for building a dataset of readability-improving commits

Most of the commits actually improve readability



Model for improving code readability

Accuracy and BLEU in line with other coding tasks

Behavior of the input code rarely modified

Readability actually increases