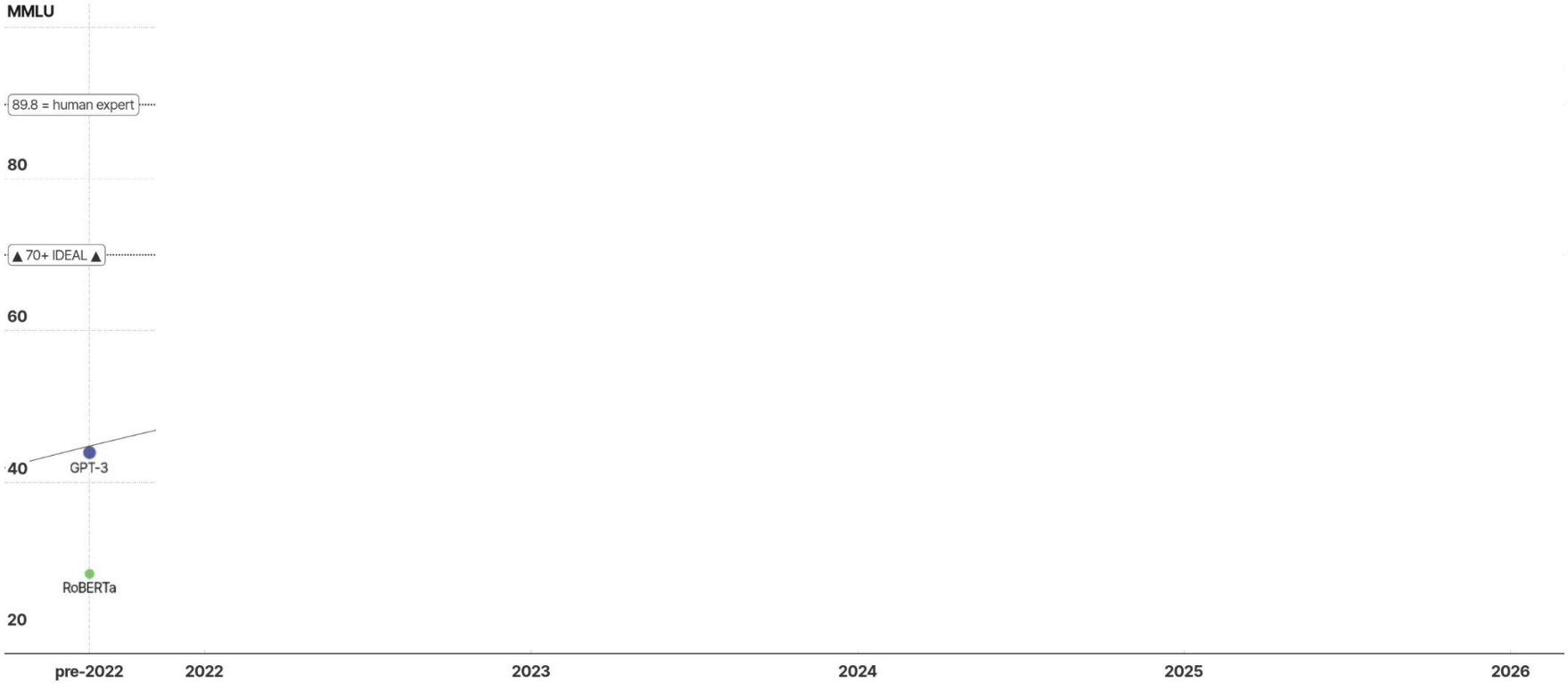
A Catalog of Data Smells for Coding Tasks

Antonio Vitale, Rocco Oliveto, Simone Scalabrino

ICSE Journal First
Transactions on Software Engineering and Methodology (TOSEM)

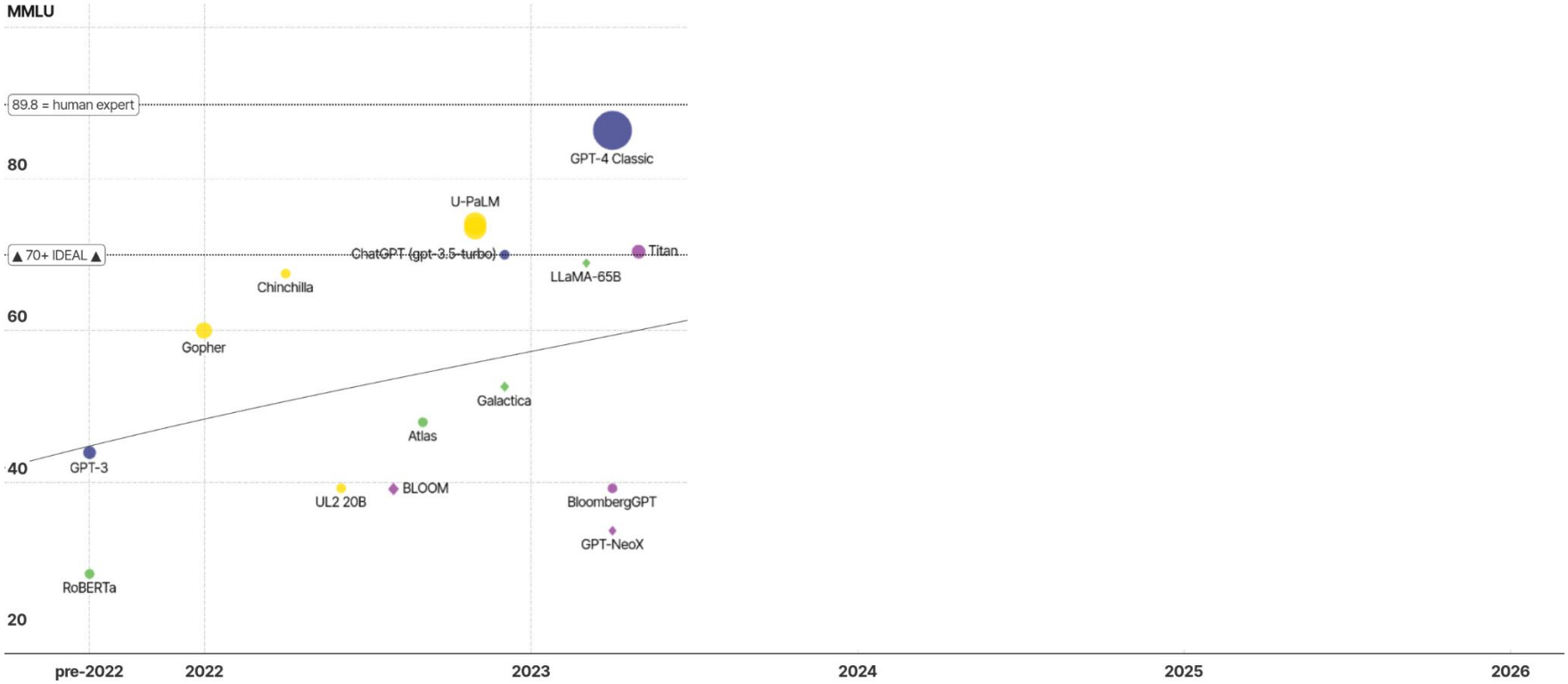
BACKGROUND



David McCandless, Tom Evans, Paul Barton
Informationisbeautiful // Jan 2026

MMLU = benchmark for measuring LLM capabilities
* = parameters undisclosed // source: LifeArchitect // data

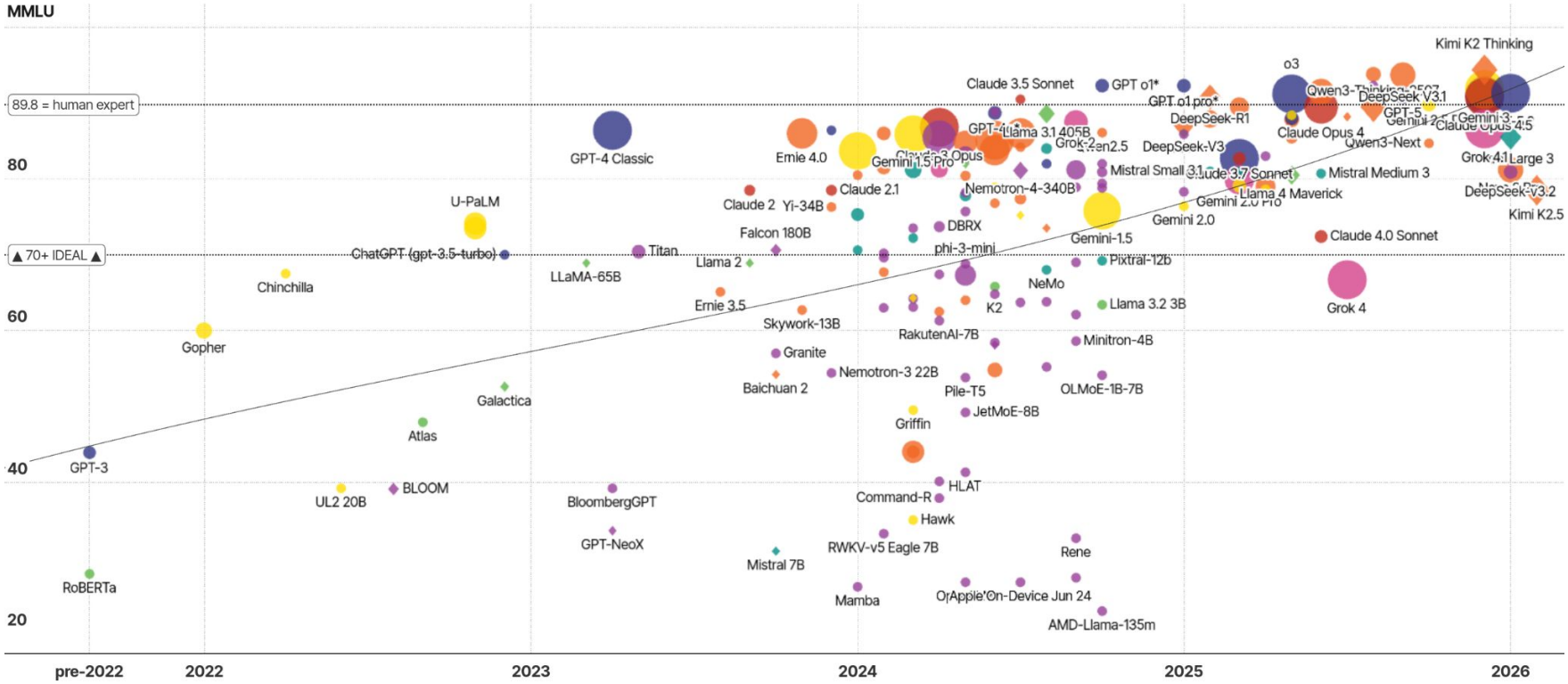
BACKGROUND



David McCandless, Tom Evans, Paul Barton
Informationisbeautiful // Jan 2026

MMLU = benchmark for measuring LLM capabilities
* = parameters undisclosed // source: LifeArchitect // data

BACKGROUND



David McCandless, Tom Evans, Paul Barton
Informationisbeautiful // Jan 2026

MMLU = benchmark for measuring LLM capabilities
* = parameters undisclosed // source: LifeArchitect // data

MADE WITH VIZsweet

BACKGROUND

CodeBERT:
A Pre-Trained Model for Programming and Natural Languages

Zhangyin Feng[1]; Daya Guo[2]; Duyu Tang[3], Nan Duan[3], Xiaocheng Feng[1]
Ming Gong[4], Linjun Shou[4], Bing Qin[1], Ting Liu[1], Daxin Jiang[4], Ming Zhou[3]
[1] Research Center for Social Computing and Information Retrieval, Harbin Institute of Technology, China
[2] The School of Data and Computer Science, Sun Yat-sen University, China
[3] Microsoft Research Asia, Beijing, China
[4] Microsoft Search Technology Center Asia, Beijing, China
{zyfeng,xcfeng,qinb,tliu}@ir.hit.edu.cn
guody5@mail2.sysu.edu.cn
{dutang,nanduan,migon,lisho,djiang,mingzhou}@microsoft.com

Abstract

We present CodeBERT, a *bimodal* pre-trained model for programming language (PL) and natural language (NL). CodeBERT learns general-purpose representations that support downstream NL-PL applications such as natural language code search, code documentation generation, etc. We develop CodeBERT with Transformer-based neural architecture, and train it with a hybrid objective function that incorporates the pre-training task of replaced token detection, which is to detect plausible alternatives sampled from generators. This enables us to utilize both "*bimodal*" data of NL-PL pairs and "*unimodal*" data, where the former provides input tokens for model training while the latter helps to learn better generators. We evaluate CodeBERT on two NL-PL applications by fine-tuning model parameters. Results show that CodeBERT achieves state-of-the-art performance on both natural language code search and code documentation generation. Furthermore, to investigate what type of knowledge is learned in CodeBERT, we construct a dataset for NL-PL probing, and evaluate in a zero-shot setting where parameters of pre-trained models are fixed. Results show that CodeBERT performs better than previous pre-trained models on NL-PL probing.

1 Introduction

Large pre-trained models such as ELMo (Peters et al., 2018), GPT (Radford et al., 2018), BERT (Devlin et al., 2018), XLNet (Yang et al., 2019) and RoBERTa (Liu et al., 2019) have dramatically improved the state-of-the-art on a variety of natural language processing (NLP) tasks. These pre-trained models learn effective contextual representations from massive unlabeled text optimized by self-supervised objectives, such as masked language modeling, which predicts the original masked word from an artificially masked input sequence. The success of pre-trained models in NLP also drives a surge of multi-modal pre-trained models, such as ViLBERT (Lu et al., 2019) for language-image and VideoBERT (Sun et al., 2019) for language-video, which are learned from *bimodal* data such as language-image pairs with *bimodal* self-supervised objectives.

In this work, we present CodeBERT, a *bimodal* pre-trained model for natural language (NL) and programming language (PL) like Python, Java, JavaScript, etc. CodeBERT captures the semantic connection between natural language and programming language, and produces general-purpose representations that can broadly support NL-PL understanding tasks (e.g. natural language code search) and generation tasks (e.g. code documentation generation). It is developed with the multi-layer Transformer (Vaswani et al., 2017), which is adopted in a majority of large pre-trained models. In order to make use of both *bimodal* instances of NL-PL pairs and large amount of available *unimodal* codes, we train CodeBERT with a hybrid objective function, including standard masked language modeling (Devlin et al., 2018) and replaced token detection (Clark et al., 2020), where *unimodal* codes help to learn better generators for producing better alternative tokens for the latter objective.

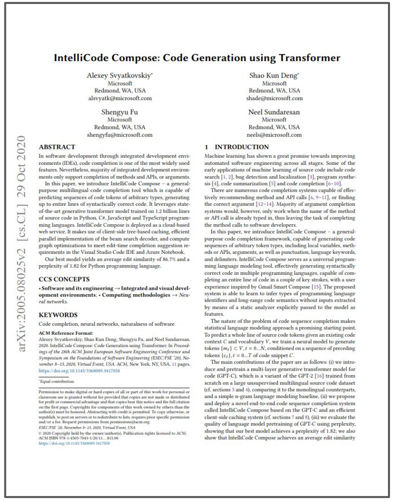
We train CodeBERT from Github code reposito-

[*] Work done while this author was an intern at Microsoft Research Asia.

[†] All the codes and data are available at https://github.com/microsoft/CodeBERT

2020

BACKGROUND



2020

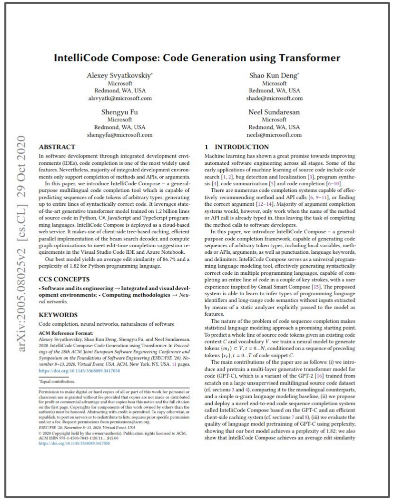
2020

BACKGROUND

CodeBERT:
A Pre-Trained Model for Programming and Natural Languages

Zhangyin Feng[1]; Daya Guo[2]; Duyu Tang[3], Nan Duan[3], Xiaocheng Feng[1]
Ming Gong[4], Linjun Shou[4], Bing Qin[1], Ting Liu[1], Daxin Jiang[4], Ming Zhou[3]
[1] Research Center for Social Computing and Information Retrieval, Harbin Institute of Technology, China
[2] The School of Data and Computer Science, Sun Yat-sen University, China
[3] Microsoft Research Asia, Beijing, China
[4] Microsoft Search Technology Center Asia, Beijing, China

arXiv:2002.08155v4 [cs.CL] 18 Sep 2020



IntelliCode Compose: Code Generation using Transformer

Alexey Svyatkovskiy, Microsoft, Redmond, WA, USA
Shao Kun Deng, Microsoft, Redmond, WA, USA
Shengyu Fu, Microsoft, Redmond, WA, USA
Neel Sundaresan, Microsoft, Redmond, WA, USA

arXiv:2005.08025v2 [cs.CL] 29 Oct 2020

A Transformer-based Approach for Source Code Summarization

Wasi Uddin Ahmad, University of California, Los Angeles
Saikat Chakraborty, Columbia University
Baishakhi Ray, Columbia University
Kai-Wei Chang, University of California, Los Angeles

arXiv:2005.00653v1 [cs.SE] 1 May 2020

2020

2020

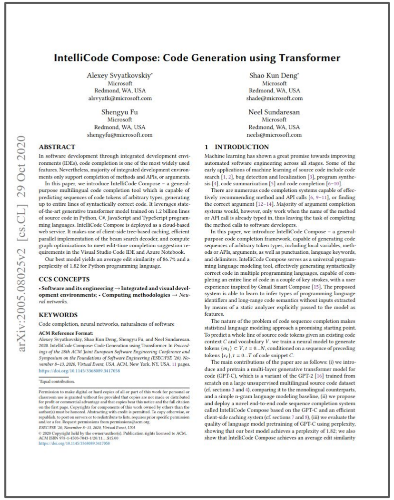
2020

BACKGROUND

CodeBERT: A Pre-Trained Model for Programming and Natural Languages

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, Ming Zhou

arXiv:2002.08155v4 [cs.CL] 18 Sep 2020



IntelliCode Compose: Code Generation using Transformer

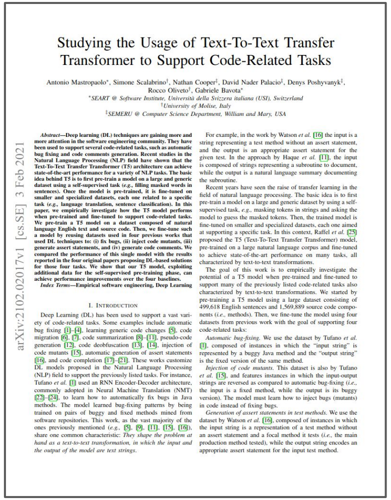
Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, Neel Sundaresan

arXiv:2005.08025v2 [cs.CL] 29 Oct 2020

A Transformer-based Approach for Source Code Summarization

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, Kai-Wei Chang

arXiv:2005.00653v1 [cs.SE] 1 May 2020



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

arXiv:2102.02017v1 [cs.SE] 3 Feb 2021

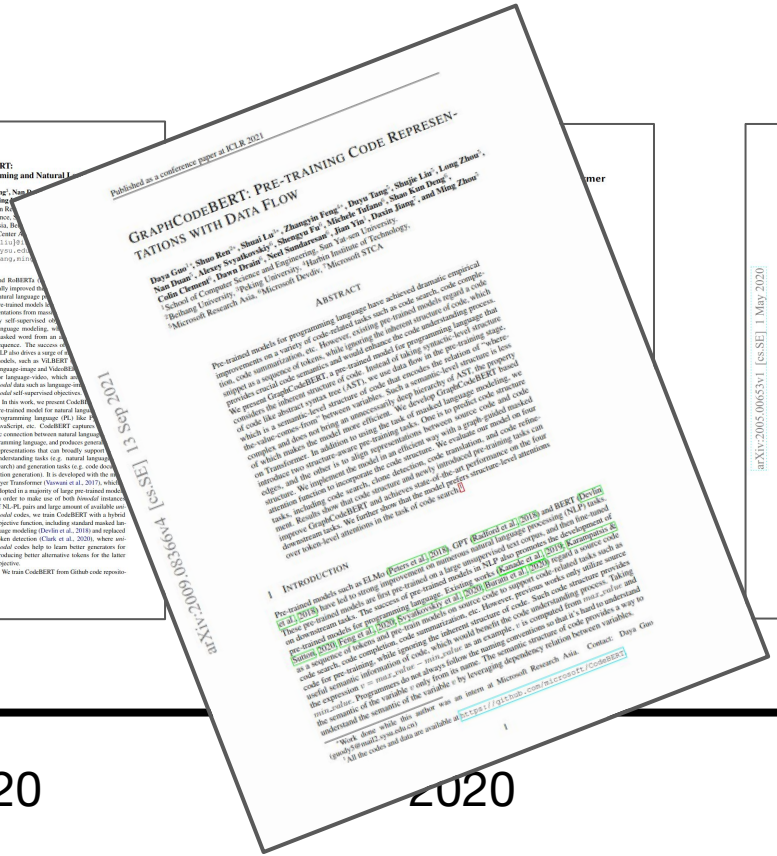
2020

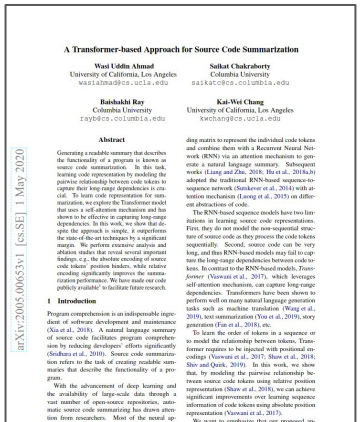
2020

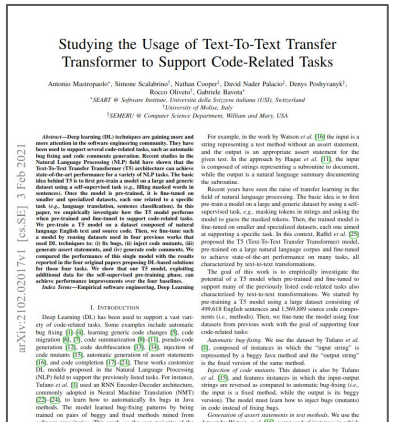
2020

2021

BACKGROUND







2020 2020 2020 2021

BACKGROUND

2020 2020 2020 2021

BACKGROUND

2020 2021

BACKGROU

VulRepair: A T5-Based Automated Software Vulnerability Repair

Michael Fu
yeh.fu@monash.edu
Monash University
Australia

Chakkrit Tantithamthavorn*
chakkrit@monash.edu
Monash University
Australia

Trung Le
trunglm@monash.edu
Monash University
Australia

Van Nguyen
khacvan.nguyen@adelaide.edu.au
The University of Adelaide
Australia

Dinh Phung
dinh.phung@monash.edu
Monash University
Australia

Repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549098

ABSTRACT

As software vulnerabilities grow in volume and complexity, researchers proposed various Artificial Intelligence (AI)-based approaches to help under-resourced security analysts to find, detect, and localize vulnerabilities. However, security analysts still have to spend a huge amount of effort to manually fix or repair such vulnerable functions. Recent work proposed an NMT-based Automated Vulnerability Repair, but it is still far from perfect due to various limitations. In this paper, we propose VulRepair, a T5-based automated software vulnerability repair approach that leverages the pre-training and BPE components to address various technical limitations of prior work. Through an extensive experiment with over 8,482 vulnerability fixes from real-world software projects, we find that our VulRepair achieves a Perfect Prediction of 44%, which is 13%–21% more accurate than competitive baseline approaches. These results lead us to conclude that our VulRepair is considerably more accurate than two baseline approaches, highlighting the substantial advancement of NMT-based Automated Vulnerability Repairs. Our additional investigation also shows that our VulRepair can accurately repair as many as 745 out of 1,706 real-world well-known vulnerabilities (e.g., Use After Free, Improper Input Validation, OS Command Injection), demonstrating the practicality and significance of our VulRepair for generating vulnerability repairs, helping under-resourced security analysts on fixing vulnerabilities.

CCS CONCEPTS

• **Software and its engineering**; • **Security and privacy**;

KEYWORDS

Software Vulnerability Repair

ACM Reference Format:

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability

*The corresponding author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00

1 INTRODUCTION

Software vulnerabilities are security flaws, glitches, or weaknesses found in software systems that could be exploited by attackers to undertake malicious activities [6]. In particular, criminal groups may make use of unresolved security vulnerabilities in software to attack and damage a system to steal confidential information or extort assets, resulting in severe economic damage [17]. According to the statistics of the National Vulnerability Database (NVD), the number of vulnerabilities discovered per year is considerably increased five times from 4k+/year in 2011 to 20k+/year in 2021 [1]. Recently, researchers proposed various Artificial Intelligence (AI)-based approaches to help under-resourced security analysts better understand the characteristics of vulnerabilities (e.g., vulnerability analysis) [2, 3, 5, 7, 44, 57, 58, 62] and find vulnerabilities faster (e.g., vulnerability predictions) [12, 20, 21, 25, 26, 34, 37, 43, 49–51, 55, 63, 65]. For example, AI-based vulnerability prediction approaches are proposed to predict if a given function is vulnerable or not at the various granularity levels (e.g., function [12, 37, 43, 65], line [20, 21, 28, 36, 38–40, 48, 59]) using various types of information (e.g., graph properties [12, 21, 55, 65], and mutual information [36]). Such vulnerability prediction approaches only help security analysts to find, detect, and localize the location of vulnerabilities. However, security analysts still have to spend a huge amount of effort to manually fix or repair vulnerabilities [4, 11, 35].

Recently, Chen *et al.* [13] proposed VRepair [13], an automated vulnerability repair approach that leverages a Transformer-based Neural Machine Translation (NMT). VRepair is proposed to address various challenges of prior work in the area of automated program repairs (e.g., SequenceR [14], an RNN-based NMT model). However, VRepair is still inaccurate due to the following three limitations.

Limitation ①: VRepair is trained on a small bug-fix corpus of 23,607 C/C++ functions, which may generate suboptimal vector representations.

Limitation ②: VRepair leverages the word-level tokenization and the copy mechanism to handle Out-Of-Vocabulary (OOV), limiting its ability to generate new tokens that never appear in a vulnerable function but are newly introduced in the vulnerability repair.

BACKGROU

VulRepair: A T5-Based Automated Software Vulnera

Michael Fu
yeh.fu@monash.edu
Monash University
Australia

Chakkrit Tantithamthavorn*
chakkrit@monash.edu
Monash University
Australia

Van Nguyen
khacvan.nguyen@adelaide.edu.au
The University of Adelaide
Australia

ABSTRACT

As software vulnerabilities grow in volume and complexity, researchers have proposed various Artificial Intelligence (AI)-based approaches to help under-resourced security analysts to find, detect, and localize vulnerabilities. However, security analysts still have to spend a huge amount of effort to manually fix or repair such vulnerable functions. Recent work proposed an NMT-based Automated Vulnerability Repair, but it is still far from perfect due to various limitations. In this paper, we propose VulRepair, a T5-based automated software vulnerability repair approach that leverages the pre-training and BPE components to address various technical limitations of prior work. Through an extensive experiment with over 8,482 vulnerability fixes from real-world software projects, we find that our VulRepair achieves a Perfect Prediction of 44%, which is 13%–21% more accurate than competitive baseline approaches. These results lead us to conclude that our VulRepair is considerably more accurate than two baseline approaches, highlighting the substantial advancement of NMT-based Automated Vulnerability Repairs. Our additional investigation also shows that our VulRepair can accurately repair as many as 745 out of 1,706 real-world vulnerabilities (e.g., Use After Free, Improper Validation, OS Command Injection), demonstrating the practical significance of our VulRepair for generating vulnerability fixes, helping under-resourced security analysts on fixing v

CCS CONCEPTS

• **Software and its engineering** • **Security an**

KEYWORDS

Software Vulnerability Repair

ACM Reference Format:

Michael Fu, Chakkrit Tantithamthavorn, Tr
Phung. 2022. VulRepair: A T5-Based Aut

*The corresponding author

SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics

He Ye
heye@kth.se
KTH Royal Institute of Technology
Sweden

Matias Martinez
matias.martinez@uphf.fr
Université Polytechnique Hauts-de-France
France

Xiapu Luo
csxluo@comp.polyu.edu.hk
The Hong Kong Polytechnic University
China

Tao Zhang
tazhang@must.edu.mo
Macau University of Science and Technology
China

Martin Monperrus
monperrus@kth.se
KTH Royal Institute of Technology
Sweden

ABSTRACT

Learning-based program repair has achieved good results in a recent series of papers. Yet, we observe that the related work fails to repair some bugs because of a lack of knowledge about 1) the application domain of the program being repaired, and 2) the fault type being repaired. In this paper, we solve both problems by changing the learning paradigm from supervised training to self-supervised training in an approach called SelfAPR. First, SelfAPR generates training samples on disk by perturbing a previous version of the program being repaired, enforcing the neural model to capture project-specific knowledge. This is different from the previous work based on mined past commits. Second, SelfAPR executes all training samples and extracts and encodes test execution diagnostics into the input representation, steering the neural model to fix the kind of fault. This is different from the existing studies that only consider static source code as input. We implement SelfAPR and evaluate it in a systematic manner. We generate 1 039 873 training samples obtained by perturbing 17 open-source projects. We evaluate SelfAPR on 818 bugs from Defects4J. SelfAPR correctly repairs 110 of them, outperforming all the supervised learning repair approaches.

ACM Reference Format:

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556926

1 INTRODUCTION

Automated program repair (APR) aims to reduce the manual and costly work related to the bug localization and bug fixing tasks of software maintenance [22, 49, 51]. While early works in the field mainly used search-based [36, 37] or semantics-based [47,

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556926

Figure 1: Motivating example of SelfAPR: supervised neural program repair is fundamentally limited by the absence of project-specific tokens at training time.

60] techniques, recently, a different line APR research has proven successful: neural machine translation for program repair, or simply "neural program repair" [10, 11, 13, 42, 68, 83, 85].

Neural program repair is based on a common encoder-decoder architecture to transform the buggy code to the correct code, yet the proposed approaches differ as follows: 1) In the input or output representations, for example, the decoders may output code edits [85]; 2) In the pre-processing or post-processing phases of the data, for example filtering the patches with invalid identifiers [31]. When those past works are compared one against the others on the same benchmark [83], those variations explain the performance differences.

Despite those differences, the previous work on neural program repair is dominantly founded on the same machine learning paradigm: supervised learning [26]. In that context, the supervised training samples come from mining real-world commits made by human developers. For example, Recoder's [85] training samples were downloaded from GitHub, totaling 1 083 185 commits done between March 2011 and March 2018. In this paper, we claim and provide evidence that this supervised paradigm for neural program repair has two fundamental limitations as follows.

Problem 1: Lack of project-specific knowledge. Past commits used for training are typically collected from open-source projects. Those projects may have little or nothing to do with the application domain of the program under repair. This is known as the exposure bias problem [5]. In other terms, the training samples do not contain project-specific knowledge. By project-specific knowledge, we refer to specific fixing tokens of variables, expressions and statements, and their semantic relationships that the neural network can use to

Studying the Usage of Text-To-Text Transfer … Support Code-Related Tasks

…e Usage of Transformer …de Completion

2021

BACKGROUND

VulRepair: A T5-Based Automated Software Vulnera
Michael Fu
yeh.fu@monash.edu
Monash University
Australia
Chakkrit Tantithamthavorn*
chakkrit@monash.edu
Monash University
Australia
Van Nguyen
khacvan.nguyen@adelaide.edu.au
The University of Adelaide
Australia
Dinh Phung
ABSTRACT
CCS CONCEPTS
• Software and its engineering; • Security an
KEYWORDS
Software Vulnerability Repair
SelfAPR: Self-supervised Program R
with Test Execution Diagnosti
He Ye
heye@kth.se
KTH Royal Institute of Technology
Sweden
Matias Martinez
matias.martinez@uphf.fr
Université Polytechnique
Hauts-de-France
France
Tao Zhang
tazhang@must.edu.mo
Macau University of Science and
Technology
China
Martin Mo
KTH Royal Institu
ABSTRACT
Learning-based program repair has achieved good results in a recent series of papers. Yet, we observe that the related work fails to repair some bugs because of a lack of knowledge about 1) the application domain of the program being repaired, and 2) the fault type being repaired. In this paper, we solve both problems by changing the learning paradigm from supervised training to self-supervised training in an approach called SelfAPR. First, SelfAPR generates training samples on disk by perturbing a previous version of the program being repaired, enforcing the neural model to capture project-specific knowledge. This is different from the previous work based on mined past commits. Second, SelfAPR executes all training samples and extracts and encodes test execution diagnostics into the input representation, steering the neural model to fix the kind of fault. This is different from the existing studies that only consider static source code as input. We implement SelfAPR and evaluate it in a systematic manner. We generate 1 039 873 training samples obtained by perturbing 17 open-source projects. We evaluate SelfAPR on 818 bugs from Defects4J, SelfAPR correctly repairs 110 of them, outperforming all the supervised learning repair approaches.
1 INTRODUCTION
Using Pre-Trained Models to Boost Code Review Automation
Rosalia Tufano
Luca Pascarella
Simone Masiero
Denys Poshyvanyk
Antonio Mastropaolo
Gabriele Bavota
ABSTRACT
KEYWORDS
Code Review, Empirical Study, Machine Learning on Code
1 INTRODUCTION
arXiv:2201.06850v1 [cs.SE] 18 Jan 2022


...ning to Recommend Method Names with Global Context

Fang Liu
...ab of High Confidence Software
...nology, MoE (Peking University)
Beijing, China
liufang816@pku.edu.cn

Ge Li*
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn

Zhiyi Fu
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
ypfzy@pku.edu.cn

Shuai Lu
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
lushuai96@pku.edu.cn

Yiyang Hao
Silicon Heart Tech Co., Ltd
Beijing, China
haoyiyang@nnthink.com

Zhi Jin*
Key Lab of High Confidence Software
Technology, MoE (Peking University)
Beijing, China
zhijin@pku.edu.cn

ABSTRACT

In programming, the names for the program entities, especially for the methods, are the intuitive characteristic for understanding the functionality of the code. To ensure the readability and maintainability of the programs, method names should be named properly. Specifically, the names should be meaningful and consistent with other names used in related contexts in their codebase. In recent years, many automated approaches are proposed to suggest consistent names for methods, among which neural machine translation (NMT) based models are widely used and have achieved state-of-the-art results. However, these NMT-based models mainly focus on extracting the code-specific features from the method body or the surrounding methods, the project-specific context and documentation of the target method are ignored. We conduct a statistical analysis to explore the relationship between the method names and their contexts. Based on the statistical results, we propose GTNM, a Global Transformer-based Neural Model for method name suggestion, which considers the local context, the project-specific context, and the documentation of the method simultaneously. Experimental results on java methods show that our model can outperform the state-of-the-art results by a large margin on method name suggestion, demonstrating the effectiveness of our proposed model.

CCS CONCEPTS

• Software and its engineering; • Computing methodologies → Artificial intelligence;

KEYWORDS

method name recommendation, global context, deep learning

*Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
...Association for Computing Machinery.
...4503-9221-1/22/05...$15.00
...3510154

ACM Reference Format:
Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510154

1 INTRODUCTION

During programming, developers must name variables, functions, parameters, *etc.* The appropriateness of a name changes over time during the software evolution. For example, a good function name can degrade into a poor one when the semantics of the function change or the function is used in a new context. Poor names make programs harder to understand and maintain [9, 10, 17, 22, 25, 39], leading to misuses and defects [1, 2, 8, 13]. Finding consistent names for program constructs has always been a cynosure in the software industry.

Methods are the most minor named units for indicating the program behavior in most programming languages [18], thus they are particularly important [12, 30, 32]. Meaningful and conventional method names are vital for developers to understand the behavior of programs or APIs. Once the name of a method is decided, it is laborious to change, especially when used for an API [4]. The results from an investigation in Liu et al. [28] indicate that among the change history in projects, developers usually change the method names without any change to the corresponding body code in many cases, which suggests that programmers strive to choose meaningful and appropriate method names, i.e., more consistent with other names in the same project or the codebase. Especially when collaborating, they need to obey a project's coding conventions.

In recent years, researchers have proposed automated approaches for suggesting consistent names for those methods. Based on the intuition that two methods implemented with similar code in their body code are likely to be named similarly, Liu et al. [28] proposed an IR-based approach to detect and rename inconsistent method names. They identify the inconsistent method names by comparing the names retrieved from the method body vector space with those retrieved from the method name vector space. For the inconsistent names, their model recommends the potentially consistent names by referring to the names of similarly implemented methods. However, in many cases, even the methods with similar body code can be named differently because they might belong to different projects

arXiv:2201.10705v2 [cs.S...

SelfAPR: Self-supervised Program R... with Test Execution Diagnostic...

Matias Martinez
matias.martinez@uphf.fr
Université Polytechnique
Hauts-de-France
France

Tao Zhang
...hang@must.edu.mo
...iversity of Science and
Technology
China

Martin Mo...
monperrus...
KTH Royal Institu...
Sw...

Using Pre-Trained Models to Boost Code Review Automation

Rosalia Tufano
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Luca Pascarella
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Simone Masiero
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Denys Poshyvanyk
SEMERU @ Computer Science Department
William and Mary
USA

Antonio Mastropaolo
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Gabriele Bavota
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

ABSTRACT

Code review is a practice widely adopted in open source and industrial projects. Given the non-negligible cost of such a process, researchers started investigating the possibility of automating specific code review tasks. We recently proposed Deep Learning (DL) models targeting the automation of two tasks: the first model takes as input a code submitted for review and implements in it changes likely to be recommended by a reviewer; the second takes as input the submitted code and a reviewer comment posted in natural language and automatically implements the change required by the reviewer. While the preliminary results we achieved are encouraging, both models had been tested in rather simple code review scenarios, substantially simplifying the targeted problem. This was also due to the choices we made when designing both the technique and the experiments. In this paper, we build on top of that work by demonstrating that a pre-trained Text-To-Text Transfer Transformer (T5) model can outperform previous DL models for automating code review tasks. Also, we conducted our experiments on a larger and more realistic (and challenging) dataset of code review activities.

KEYWORDS

Code Review, Empirical Study, Machine Learning on Code

1 INTRODUCTION

The benefits of code reviews have been widely recognized, with several studies providing evidence of the higher quality of reviewed code [15, 29, 31]. Also, code reviews help in preventing bugs and foster knowledge transfer among developers [10, 40]. However, studies on code reviews also highlighted an additional cost that such a process entails: Empirical evidence suggests that large software projects can undergo hundreds of code reviews per month. This applies to both open-source (e.g., ~500 reviews per month in Linux) and industrial (e.g., ~3k reviews per month in Microsoft Bing) projects. As a result, developers can spend many hours per ...iewing code [16].

Given the non-negligible cost of code review, we recently proposed the automation of specific code review tasks: The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (*i.e.,* the developer submitting the code for review) who wants to receive a rapid feedback about the code they wrote before submitting it for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_s and provide as output a revised version of C_s (*i.e.,* C_r) implementing code changes that are likely to be recommended by a reviewer.

The second scenario concerns the reviewer(s) involved in the process: a DL model is trained to take as input (i) the code C_s submitted for review, and (ii) a comment R_{nl} written by the reviewer in natural language to request a specific change on C_s. The output of the model is a revised version of C_s (*i.e.,* C_r) implementing the changes recommended in R_{nl}. The idea here is that the reviewer can use the model to provide the contributor with a concrete example of the code changes that they would like to see implemented.

In our previous work [46] we trained and experimented with the DL models on a dataset composed of ~17k triplets $\langle C_s, R_{nl}, C_r \rangle$ extracted from code reviews performed in GitHub [2] and Gerrit [1]. In particular, the model recommending code changes to the contributor is an encoder-decoder model with one encoder taking C_s as input and one decoder generating C_r. Our evaluation shows that this model can recommend a change as a reviewer would do in 3% (single prediction) to 16% of the cases (10 different predictions). The model employed in the second scenario (*i.e.,* the automated implementation of a comment recommended by the reviewer), has instead two encoders taking as input C_s and R_{nl}, respectively, and one decoder generating C_r. This model can successfully implement a change recommended by a reviewer in 12% (single prediction) to 31% (10 different predictions) of the cases.

While these results represent our first step towards automating code review tasks, our approach [46] as well as the conducted empirical study suffers of several limitations we try to overcome in this paper. First, we adopted a code abstraction process to reduce the vocabulary size and simplify the learning of the DL model. This means that the model did not work on the raw source code, but on an abstracted version of it in which, for example, variable identifiers were replaced with a special VAR_ID token, where ID is a progressive number (e.g., the second declared variable is represented by VAR_2). The possibility to go back to raw source code was guaranteed by keeping a map linking abstracted to raw ... in C_s (e.g., VAR_1 → i).

1

arXiv:2201.06850v1 [cs.SE] 18 Jan 2022

InferFix: End-to-End Program Repair with LLMs over Retrieval-Augmented Prompts

Matthew Jin
Microsoft
Redmond, WA, USA

Syed Shahriar
UCLA
Los Angeles, CA, USA

Michele Tufano
Microsoft
Redmond, WA, USA

Xin Shi
Microsoft
Redmond, WA, USA

Shuai Lu
Microsoft Research
Beijing, China

Neel Sundaresan
Microsoft
Redmond, WA, USA

Alexey Svyatkovskiy
Microsoft
Redmond, WA, USA

ABSTRACT

Software development life cycle is profoundly influenced by bugs; their introduction, identification, and eventual resolution account for a significant portion of software development cost. This has motivated software engineering researchers and practitioners to propose different approaches for automating the identification and repair of software defects.

Large language models have been adapted to the program repair task through few-shot demonstration learning and instruction prompting, treating this as an infilling task. However, these models have only focused on learning general bug-fixing patterns for uncategorized bugs mined from public repositories. In this paper, we propose InferFix: a transformer-based program repair framework paired with a state-of-the-art static analyzer to fix critical security and performance bugs. InferFix combines a Retriever – transformer encoder model pretrained via contrastive learning objective, which aims at searching for semantically equivalent bugs and corresponding fixes; and a Generator – a large language model (12 billion parameter Codex Cushman model) finetuned on supervised bug-fix data with prompts augmented via adding bug type annotations and semantically similar fixes retrieved from an external non-parametric memory.

To train and evaluate our approach, we curated InferredBugs, a novel, metadata-rich dataset of bugs extracted by executing the Infer static analyzer on the change histories of thousands of Java and C# repositories. Our evaluation demonstrates that InferFix outperforms strong LLM baselines, with a top-1 accuracy of 65.6% for generating fixes in C# and 76.8% in Java. We discuss the deployment of InferFix alongside Infer at Microsoft which offers an end-to-end solution for detection, classification, and localization of bugs, as well as fixing and validation of candidate patches, integrated in the continuous integration pipeline to automate the software development workflow.

KEYWORDS

Program repair, static analyses, prompt augmentation, finetuning

Conference'17, July 2017, Washington, DC, USA

ACM Reference Format:
Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs over Retrieval-Augmented Prompts. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

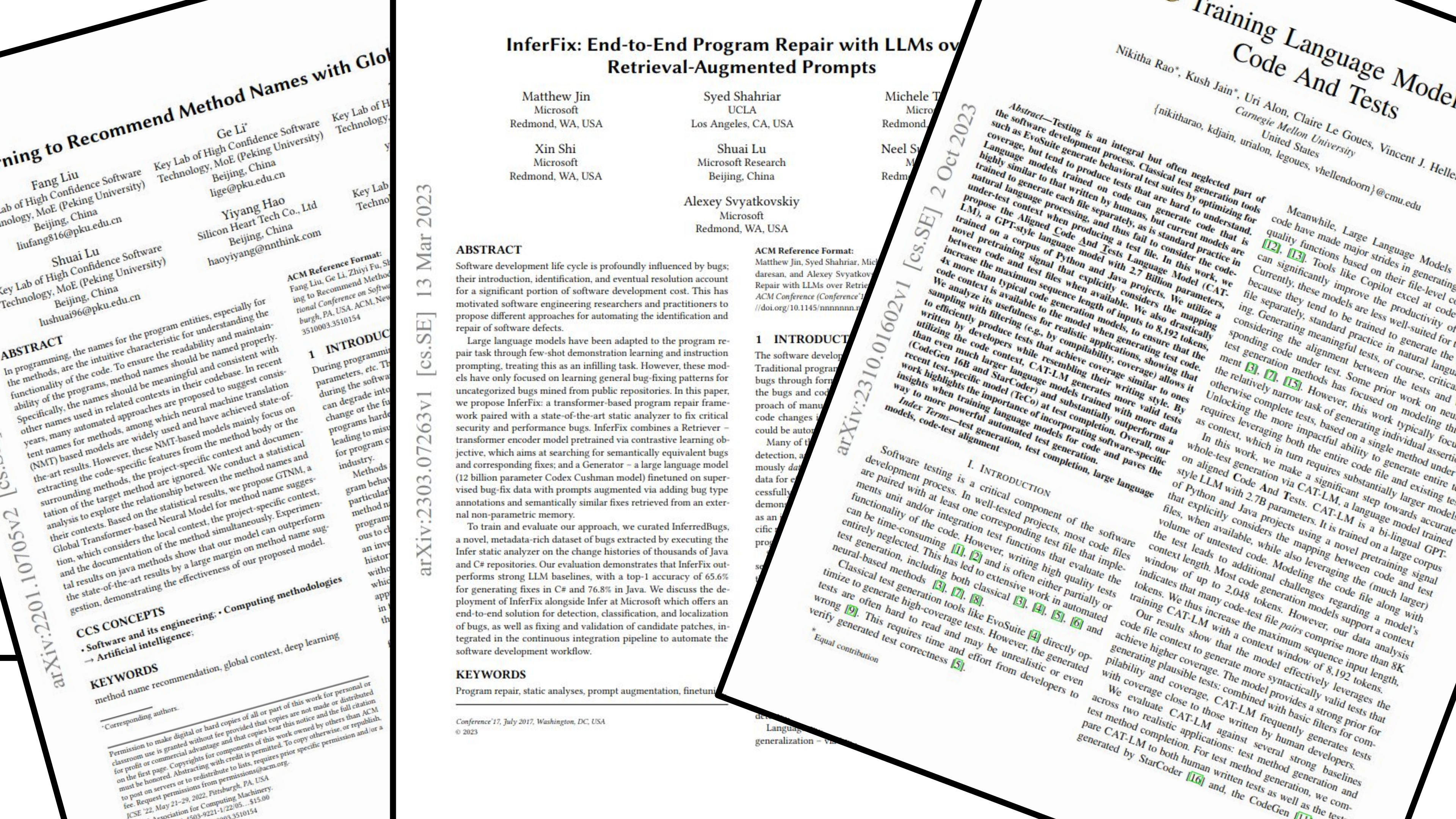
The software development lifecycle is profoundly affected by bugs. Traditional program analyses techniques can detect and localize bugs through formal reasoning, leaving the task of categorizing the bugs and coding a patch to a developer. The traditional approach of manually generating patches through examination of code changes is a time consuming and error-prone task, which could be automated.

Many of the recently proposed approaches for bug prediction, detection, and repair rely on machine learning algorithms – infamously *data hungry* – depending on large amounts of high quality data for effective training. Large language models have been successfully adapted to the program repair tasks through few-shot demonstration learning and instruction prompting, treating this as an infilling task [15]. However, while focusing on solving specific research problems they failed to provide a reliable end-to-end program repair solution that could be productized.

Static analysis tools like Infer can be used to identify critical security and performance issues. This can preempt large parts of the software development cycle, including the process of creating detailed unit tests, which can be extremely time-consuming and difficult for a large, complex project whose code is broken down into many modules or across many files. They can also identify bugs and produce bug reports in a way that is machine-readable and conducive to usage in conjunction with patch generation models.

In this work we focus on three types of bugs reported by Infer: Null Pointer Dereference (NPD), Resource Leak (RL), and Thread Safety Violation (TSV). We focus on these because they pose critical performance, reliability and security issues, and can also be more difficult to fix than other issue types which are also more commonly detected and studied.

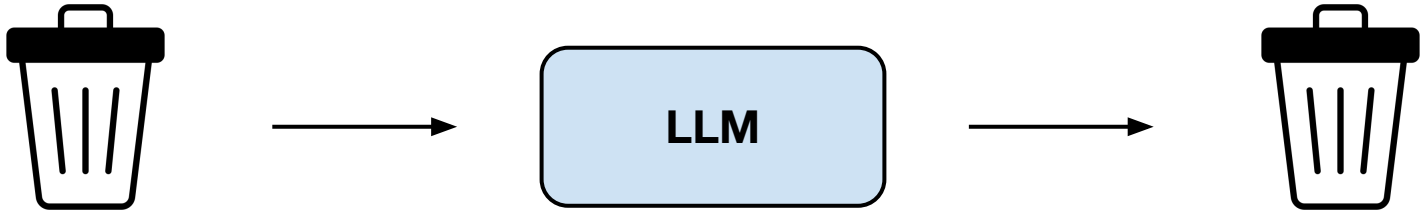
Language models commonly adopt two paradigms for task-specific generalization – via finetuning or few-shot learning. In the former



Learning to Recommend Method Names with Global Context
Fang Liu
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
liufang816@pku.edu.cn
Ge Li*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn
Shuai Lu
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lushuai96@pku.edu.cn
Yiyang Hao
Silicon Heart Tech Co., Ltd
Beijing, China
haoyiyang@nnthink.com
arXiv:2201.10705v2
ABSTRACT
In programming, the names for the program entities, especially for the methods, are the intuitive characteristic for understanding the functionality of the code. To ensure the readability and maintainability of the programs, method names should be named properly. Specifically, the names should be meaningful and consistent with other names used in related contexts in their codebase. In recent years, many automated approaches are proposed to suggest consistent names for methods, among which neural machine translation (NMT) based models are widely used and have achieved state-of-the-art results. However, these NMT-based models mainly focus on extracting the code-specific features from the method body or the surrounding methods, the project-specific context and documentation of the target method are ignored. We conduct a statistical analysis to explore the relationship between the method names and their contexts. Based on the statistical results, we propose GTNM, a Global Transformer-based Neural Model for method name suggestion, which considers the local context, the project-specific context, and the documentation of the method simultaneously. Experimental results on java methods show that our model can outperform the state-of-the-art results by a large margin on method name suggestion, demonstrating the effectiveness of our proposed model.
CCS CONCEPTS
• Software and its engineering; • Computing methodologies → Artificial intelligence;
KEYWORDS
method name recommendation, global context, deep learning
InferFix: End-to-End Program Repair with LLMs over Retrieval-Augmented Prompts
Matthew Jin
Microsoft
Redmond, WA, USA
Syed Shahriar
UCLA
Los Angeles, CA, USA
Xin Shi
Microsoft
Redmond, WA, USA
Shuai Lu
Microsoft Research
Beijing, China
Alexey Svyatkovskiy
Microsoft
Redmond, WA, USA
arXiv:2303.07263v1 [cs.SE] 13 Mar 2023
ABSTRACT
Software development life cycle is profoundly influenced by bugs; their introduction, identification, and eventual resolution account for a significant portion of software development cost. This has motivated software engineering researchers and practitioners to propose different approaches for automating the identification and repair of software defects.
Large language models have been adapted to the program repair task through few-shot demonstration learning and instruction prompting, treating this as an infilling task. However, these models have only focused on learning general bug-fixing patterns for uncategorized bugs mined from public repositories. In this paper, we propose InferFix: a transformer-based program repair framework paired with a state-of-the-art static analyzer to fix critical security and performance bugs. InferFix combines a Retriever – transformer encoder model pretrained via contrastive learning objective, which aims at searching for semantically equivalent bugs and corresponding fixes; and a Generator – a large language model (12 billion parameter Codex Cushman model) finetuned on supervised bug-fix data with prompts augmented via adding bug type annotations and semantically similar fixes retrieved from an external non-parametric memory.
To train and evaluate our approach, we curated InferredBugs, a novel, metadata-rich dataset of bugs extracted by executing the Infer static analyzer on the change histories of thousands of Java and C# repositories. Our evaluation demonstrates that InferFix outperforms strong LLM baselines, with a top-1 accuracy of 65.6% for generating fixes in C# and 76.8% in Java. We discuss the deployment of InferFix alongside Infer at Microsoft which offers an end-to-end solution for detection, classification, and localization of bugs, as well as fixing and validation of candidate patches, integrated in the continuous integration pipeline to automate the software development workflow.
KEYWORDS
Program repair, static analyses, prompt augmentation, finetuning
Conference'17, July 2017, Washington, DC, USA
© 2023
Training Language Models on Code And Tests
Nikitha Rao*, Kush Jain*, Uri Alon, Claire Le Goues, Vincent J. Hellendoorn
Carnegie Mellon University
United States
{nikitharao, kdjain, urialon, legoues, vhellendoorn}@cmu.edu
arXiv:2310.01602v1 [cs.SE] 2 Oct 2023
Abstract—Testing is an integral but often neglected part of the software development process. Classical test generation tools such as EvoSuite generate behavioral test suites by optimizing for coverage, but tend to produce tests that are hard to understand. Language models trained on code can generate code that is highly similar to that written by humans, but current models are trained to generate each file separately, as is standard practice in natural language processing, and thus fail to consider the code-under-test context when producing a test file. In this work, we propose the Aligned Code And Tests Language Model (CAT-LM), a GPT-style language model with 2.7 Billion parameters, trained on a corpus of Python and Java projects. We utilize a novel pretraining signal that explicitly considers the mapping between code and test files when available. We also drastically increase the maximum sequence length of inputs to 8,192 tokens, 4x more than typical code generation models, to ensure that the code context is available to the model when generating test code. We analyze its usefulness for realistic applications, showing that sampling with filtering (e.g., by compilability, coverage) allows it to efficiently produce tests that achieve coverage similar to ones written by developers while resembling their writing style. By utilizing the code context, CAT-LM generates more valid tests than even much larger language models trained with more data (CodeGen 16B and StarCoder) and substantially outperforms a recent test-specific model (TeCo) at test completion. Overall, our work highlights the importance of incorporating software-specific insights when training language models for code and paves the way to more powerful automated test generation.
Index Terms—test generation, test completion, large language models, code-test alignment
I. INTRODUCTION
* Equal contribution

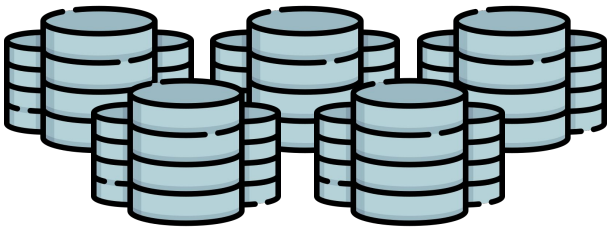
MOTIVATION

Large Language Models (LLMs) are data-driven techniques, so **“Garbage in, garbage out”** (GIGO) remains actual.



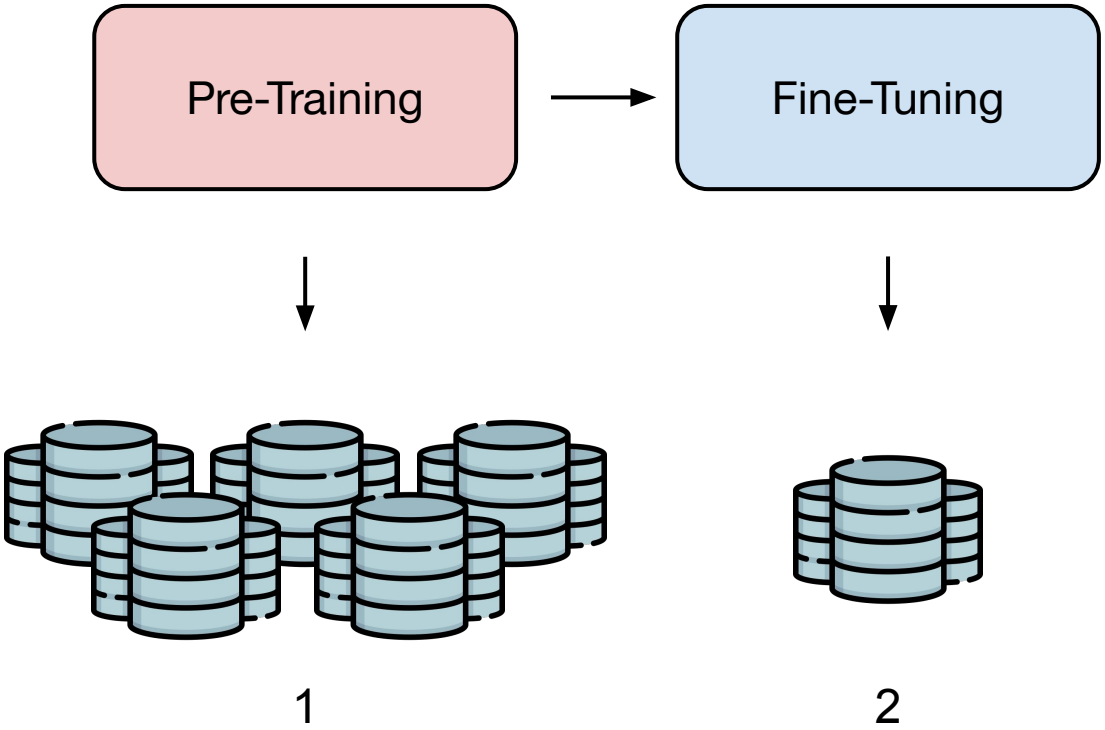
DATA IMPORTANCE

Pre-Training

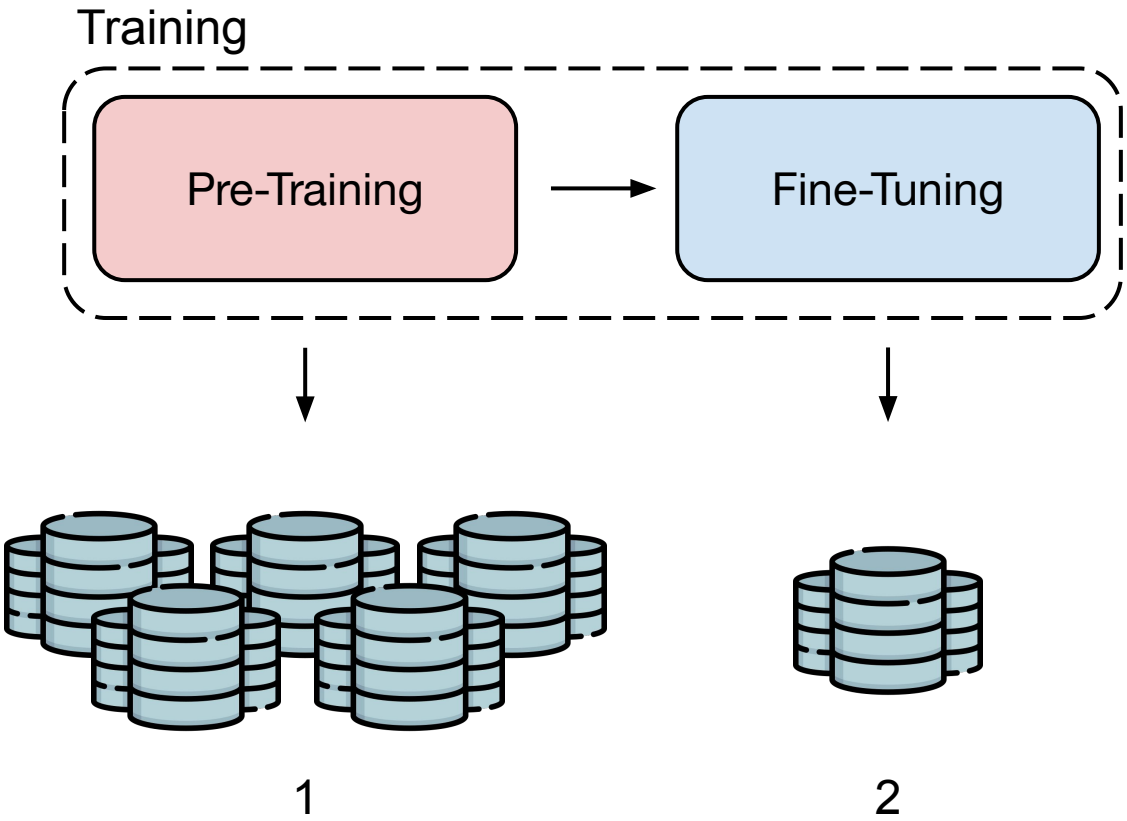


1

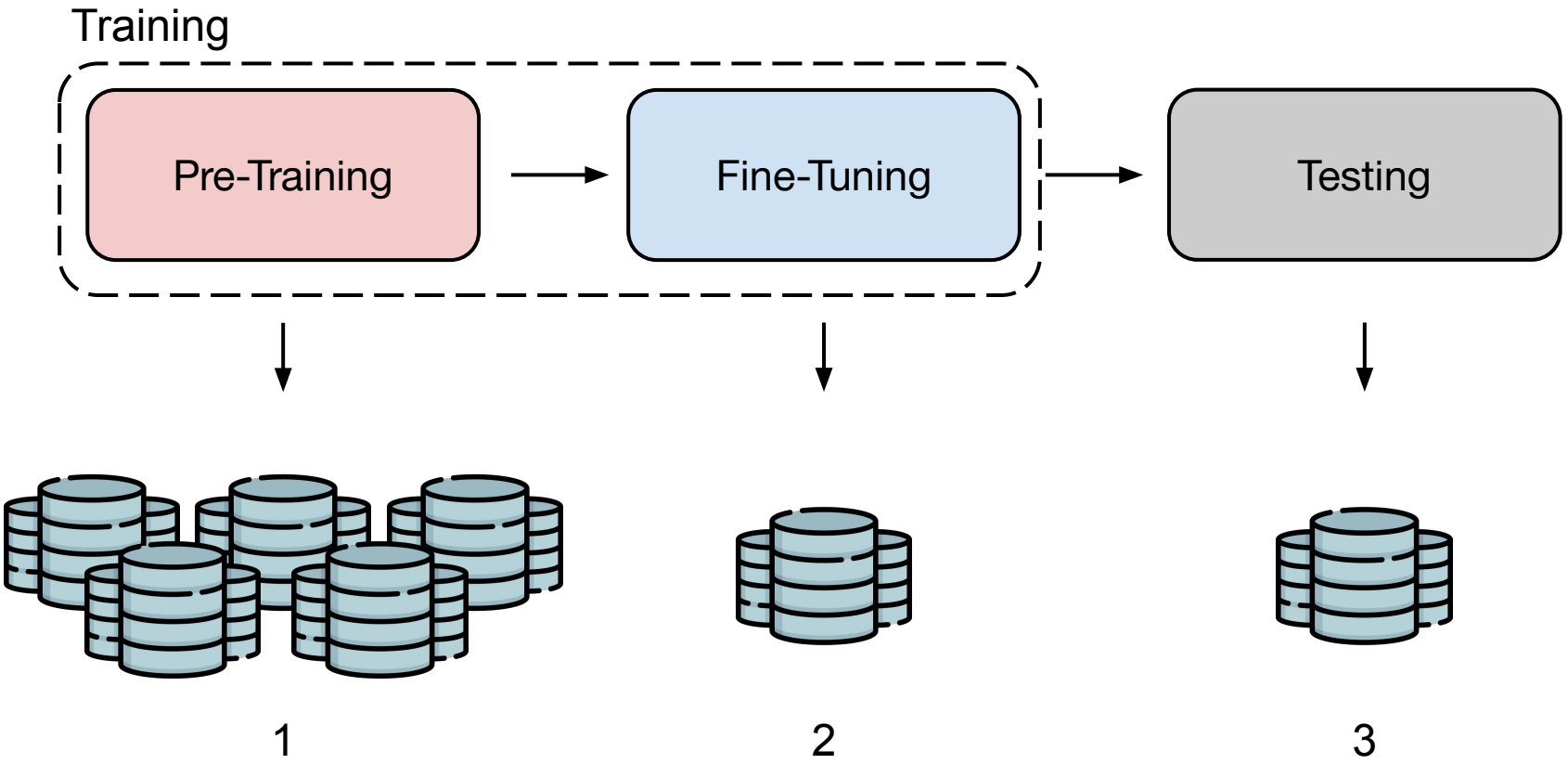
DATA IMPORTANCE



DATA IMPORTANCE



DATA IMPORTANCE

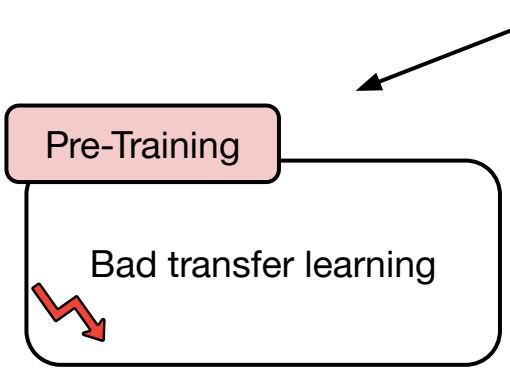


DATA IMPORTANCE

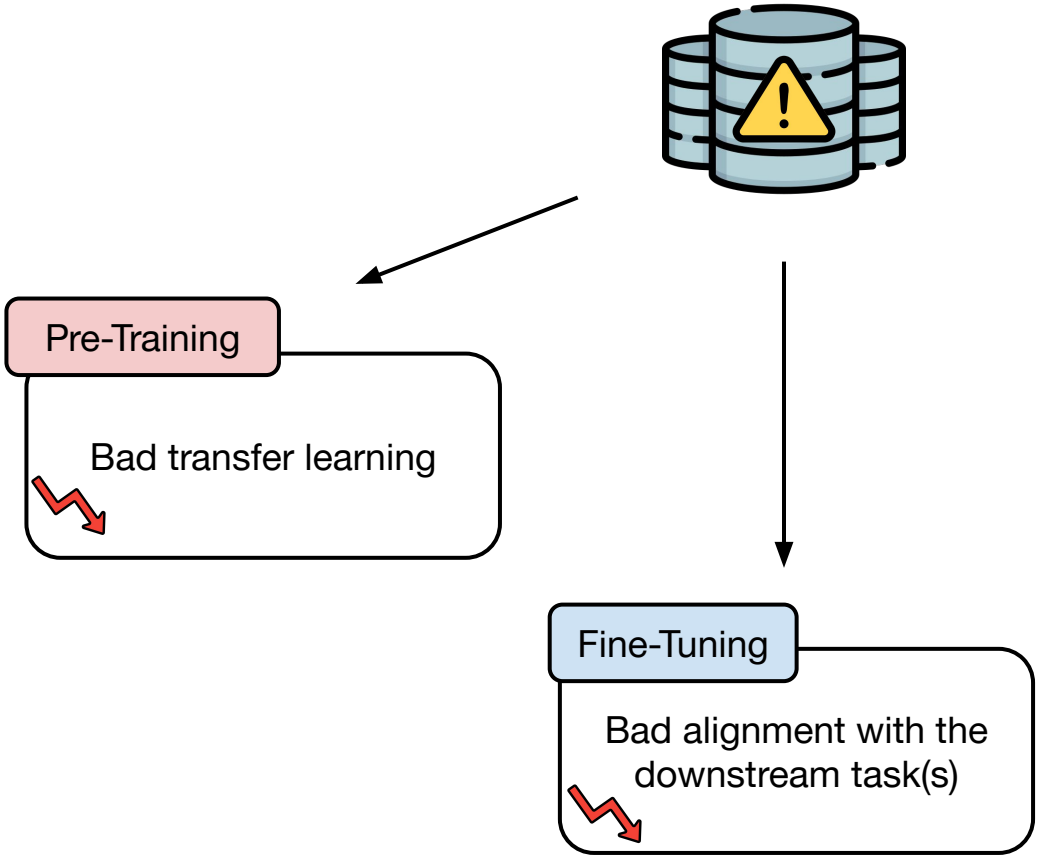
LOW-QUALITY INSTANCES

HIGH-QUALITY INSTANCES

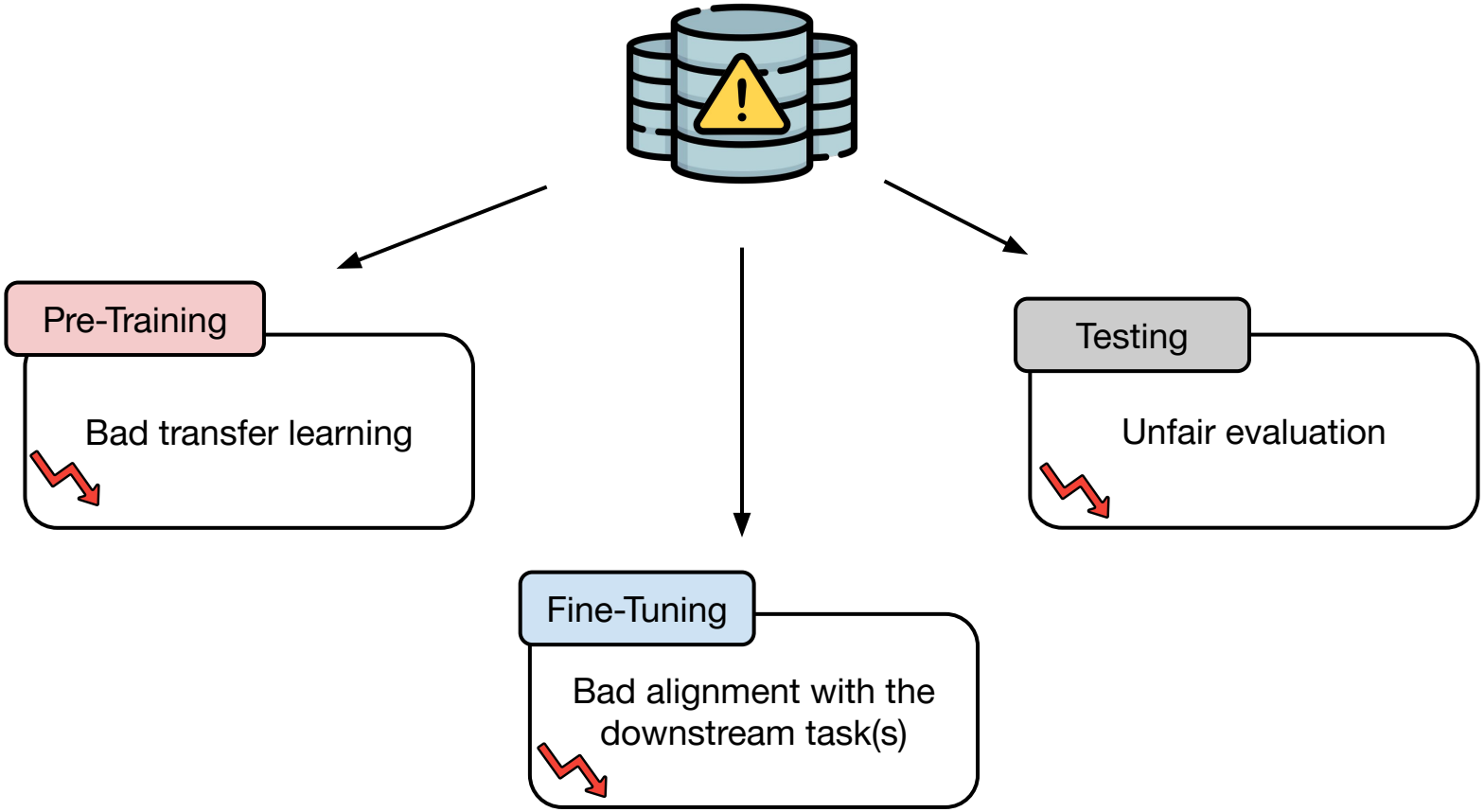
DATA IMPORTANCE



DATA IMPORTANCE



DATA IMPORTANCE

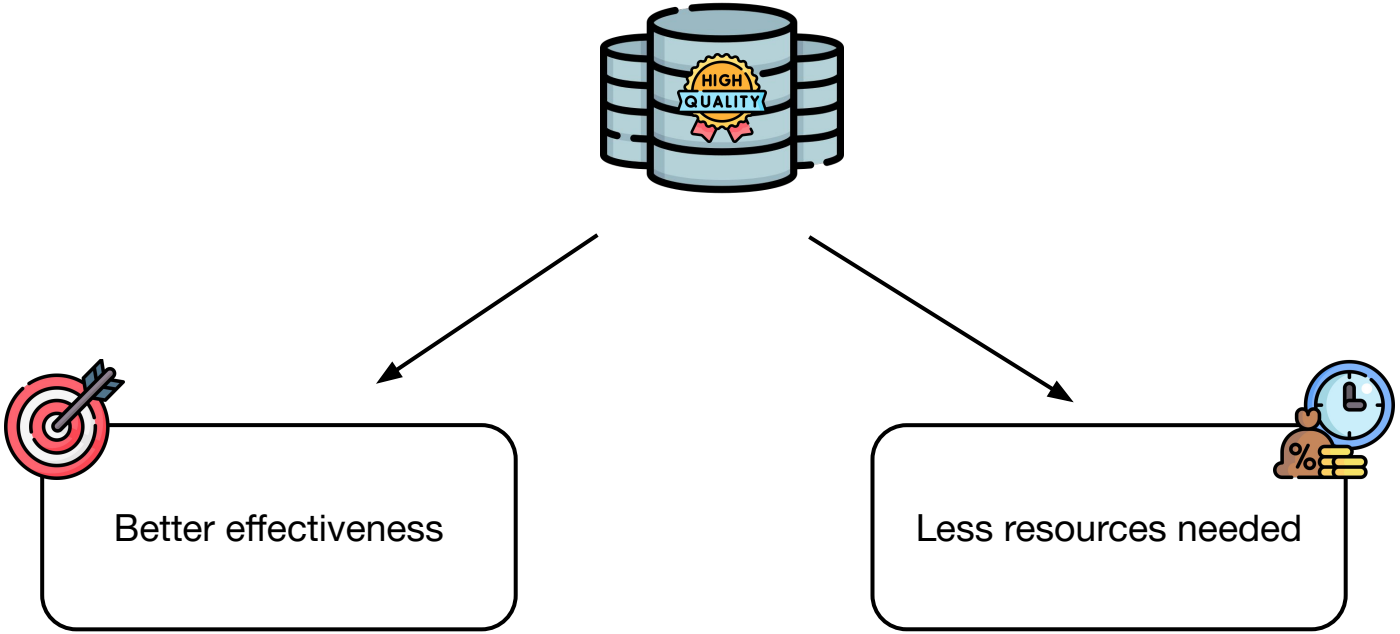


DATA IMPORTANCE

DATA IMPORTANCE

Better effectiveness

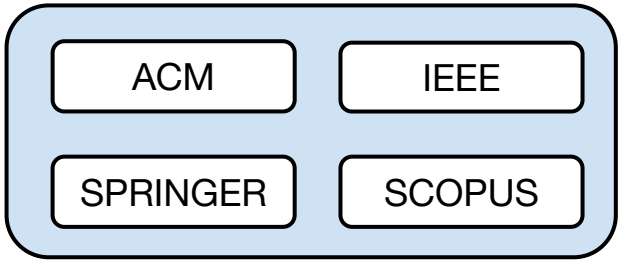
DATA IMPORTANCE



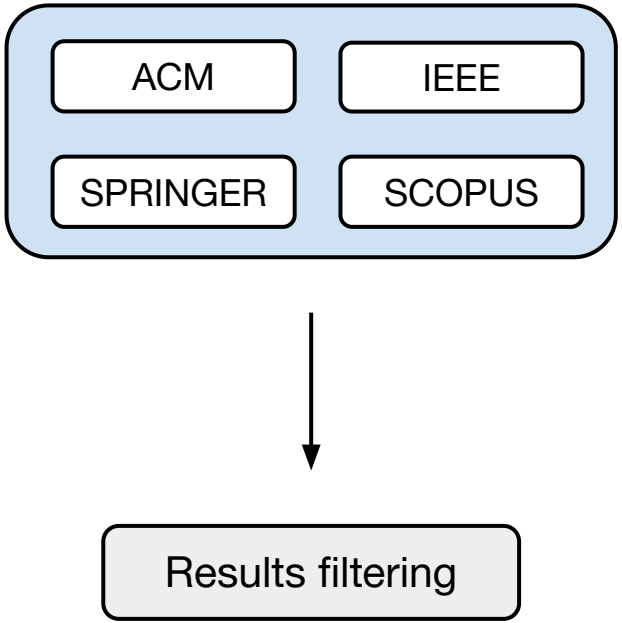
RQ1

What are the **known data smells** for code-related tasks and the **strategies** adopted to remove them?

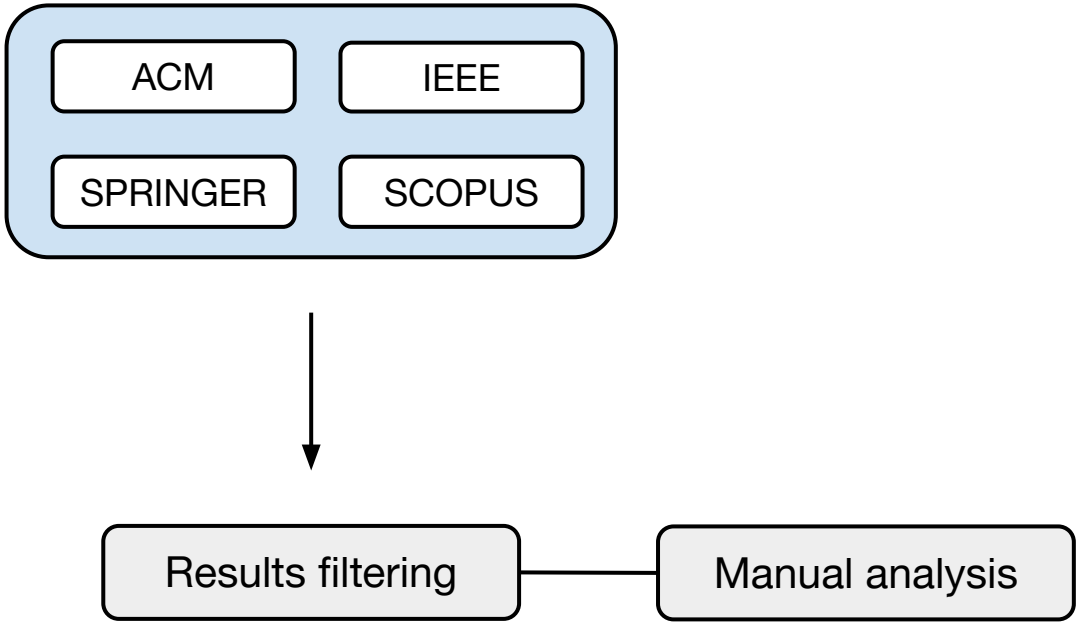
DESIGN



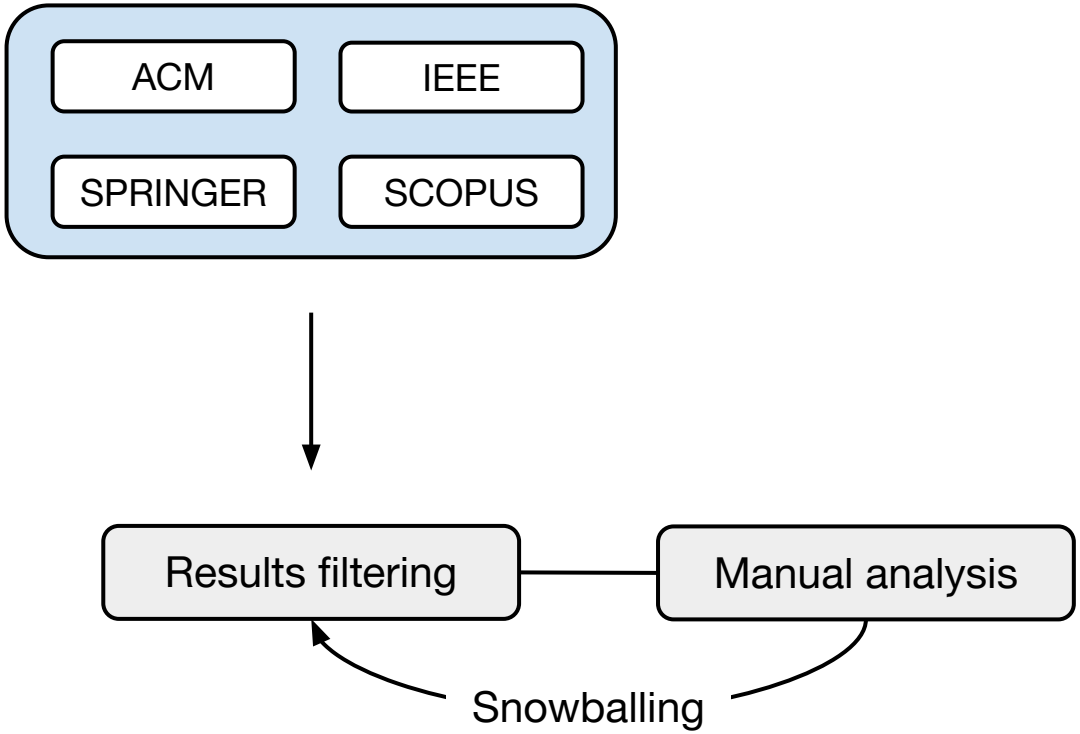
DESIGN



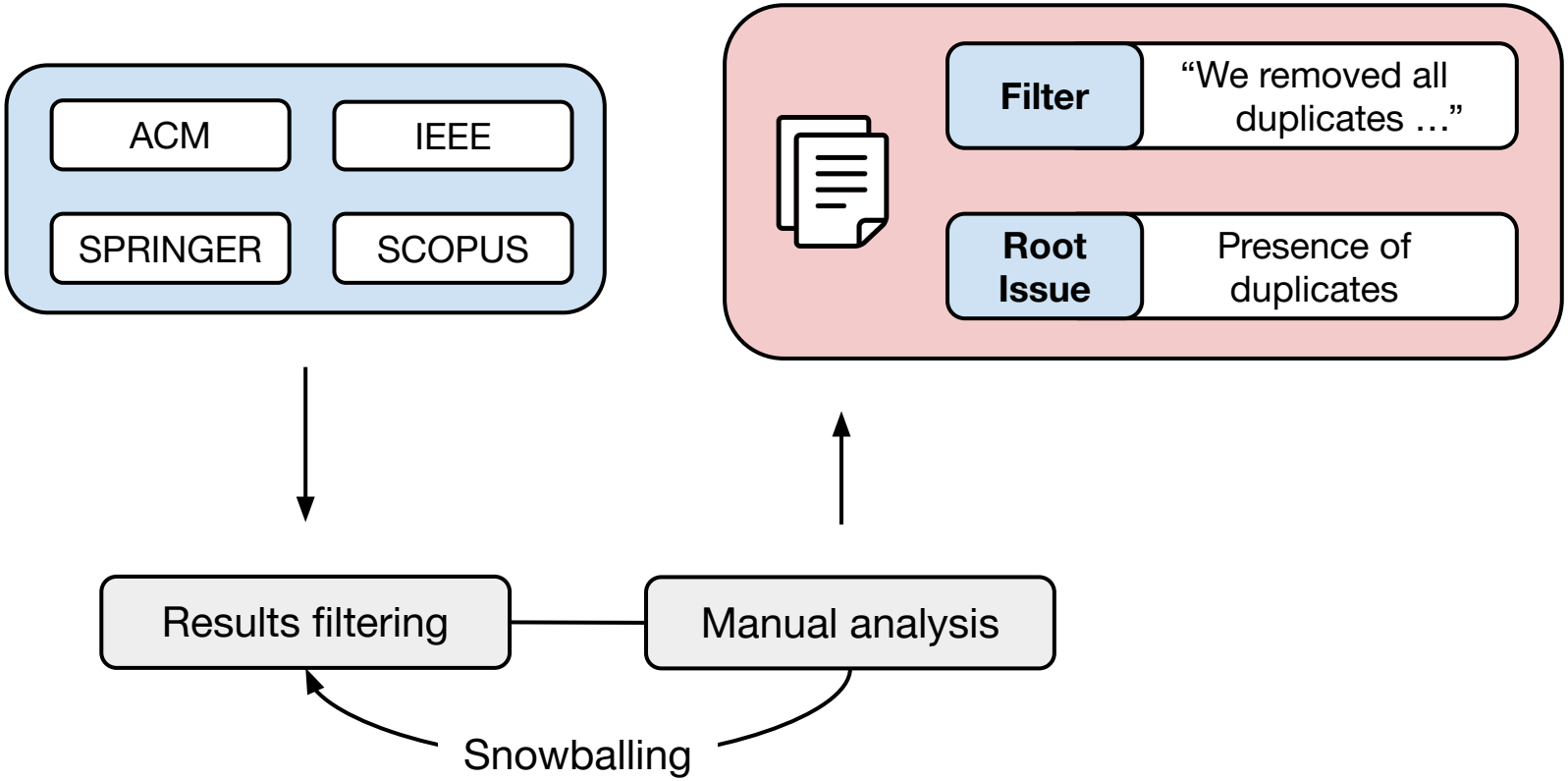
DESIGN


ACM
IEEE
SPRINGER
SCOPUS
Results filtering
Manual analysis

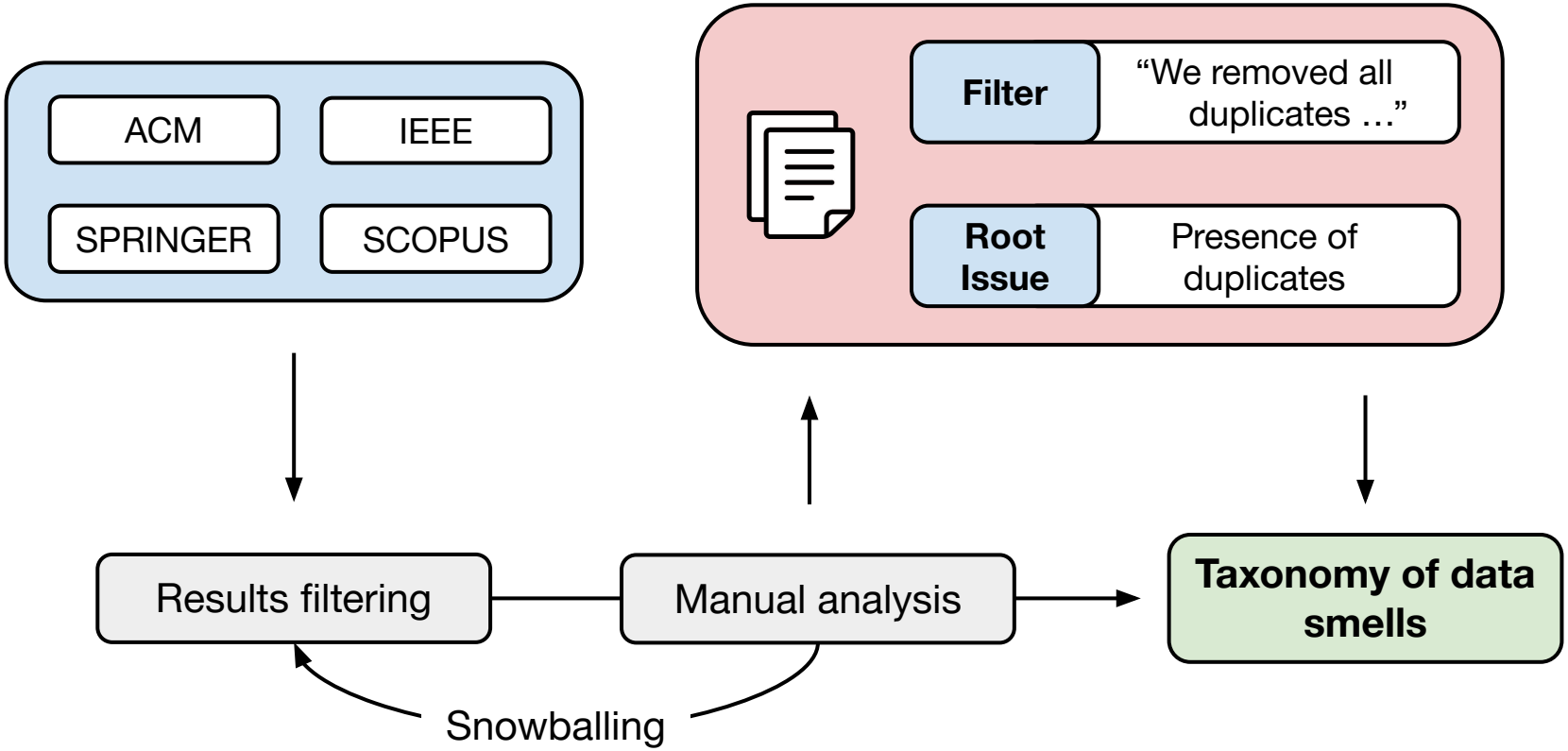
DESIGN


ACM
IEEE
SPRINGER
SCOPUS
Results filtering
Manual analysis
Snowballing

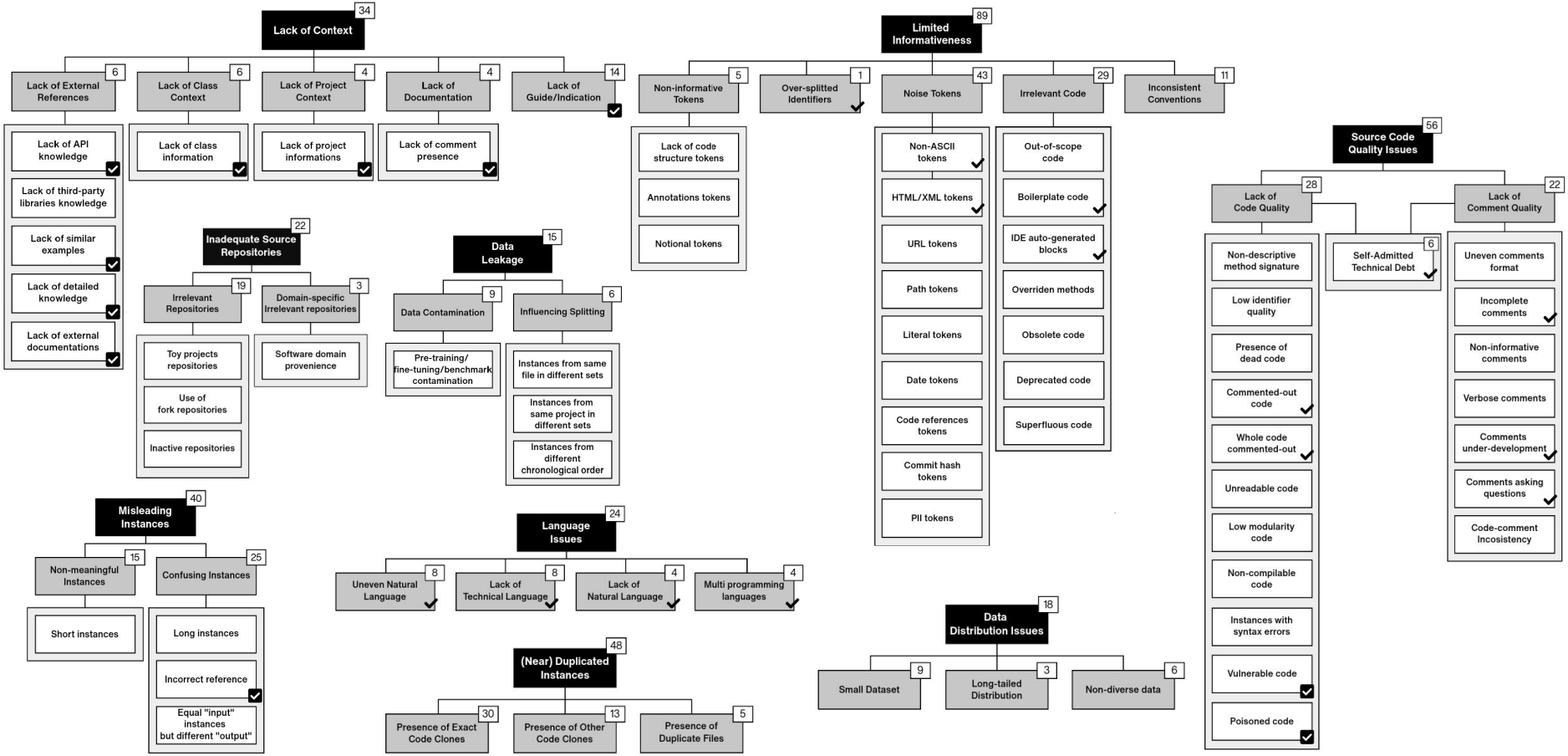
DESIGN


ACM
IEEE
SPRINGER
SCOPUS
Filter
"We removed all duplicates …"
Root Issue
Presence of duplicates
Results filtering
Manual analysis
Snowballing

DESIGN


ACM
IEEE
SPRINGER
SCOPUS
Filter
"We removed all duplicates …"
Root Issue
Presence of duplicates
Results filtering
Manual analysis
Taxonomy of data smells
Snowballing

RQ1


Lack of Context
34
Lack of External References
6
Lack of API knowledge
Lack of third-party libraries knowledge
Lack of similar examples
Lack of detailed knowledge
Lack of external documentations
Lack of Class Context
6
Lack of class information
Lack of Project Context
4
Lack of project informations
Lack of Documentation
4
Lack of comment presence
Lack of Guide/Indication
14
Limited Informativeness
89
Non-informative Tokens
5
Lack of code structure tokens
Annotations tokens
Notional tokens
Over-splitted Identifiers
1
Noise Tokens
43
Non-ASCII tokens
HTML/XML tokens
URL tokens
Path tokens
Literal tokens
Date tokens
Code references tokens
Commit hash tokens
PII tokens
Irrelevant Code
29
Out-of-scope code
Boilerplate code
IDE auto-generated blocks
Overriden methods
Obsolete code
Deprecated code
Superfluous code
Inconsistent Conventions
11
Source Code Quality Issues
56
Lack of Code Quality
28
Non-descriptive method signature
Low identifier quality
Presence of dead code
Commented-out code
Whole code commented-out
Unreadable code
Low modularity code
Non-compilable code
Instances with syntax errors
Vulnerable code
Poisoned code
Self-Admitted Technical Debt
6
Lack of Comment Quality
22
Uneven comments format
Incomplete comments
Non-informative comments
Verbose comments
Comments under-development
Comments asking questions
Code-comment Incosistency
Inadequate Source Repositories
22
Irrelevant Repositories
19
Toy projects repositories
Use of fork repositories
Inactive repositories
Domain-specific Irrelevant repositories
3
Software domain provenience
Data Leakage
15
Data Contamination
9
Pre-training/ fine-tuning/benchmark contamination
Influencing Splitting
6
Instances from same file in different sets
Instances from same project in different sets
Instances from different chronological order
Misleading Instances
40
Non-meaningful Instances
15
Short instances
Confusing Instances
25
Long instances
Incorrect reference
Equal "input" instances but different "output"
Language Issues
24
Uneven Natural Language
8
Lack of Technical Language
8
Lack of Natural Language
4
Multi programming languages
4
(Near) Duplicated Instances
48
Presence of Exact Code Clones
30
Presence of Other Code Clones
13
Presence of Duplicate Files
5
Data Distribution Issues
18
Small Dataset
9
Long-tailed Distribution
3
Non-diverse data
6

RQ1

Limited Informativeness

```
issue_title = "Fix NPE in UserService 🚨"

issue_body = """
When calling getUserById() it throws NullPointerException 😡.

Stacktrace:
java.lang.NullPointerException at com.app.service.UserService.java:42

Related commit: 9f3a1c7b2d4e8a91
Reported by: alice.dev@company.com

See details: https://github.com/org/repo/issues/123
File: /home/alice/projects/app/src/UserService.java

Note: error occurred in module — unexpected state ⚠️
"""
```

```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public String getName() {
    return name;
}
```

RQ1

Limited Informativeness

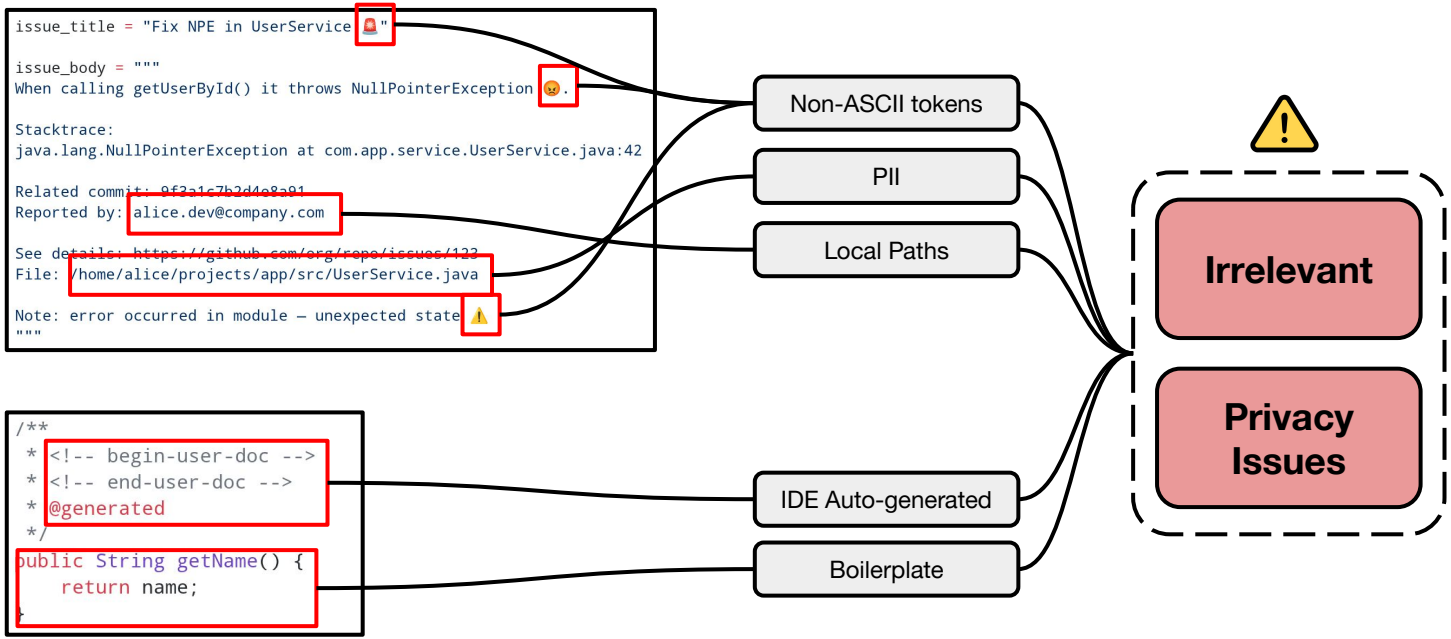
```
issue_title = "Fix NPE in UserService 🚨"

issue_body = """
When calling getUserById() it throws NullPointerException 😡.

Stacktrace:
java.lang.NullPointerException at com.app.service.UserService.java:42

Related commit: 9f3a1c7b2d4e8a01
Reported by: alice.dev@company.com

See details: https://github.com/org/repo/issues/123
File: /home/alice/projects/app/src/UserService.java

Note: error occurred in module — unexpected state ⚠️
"""
```

- Non-ASCII tokens
- PII
- Local Paths

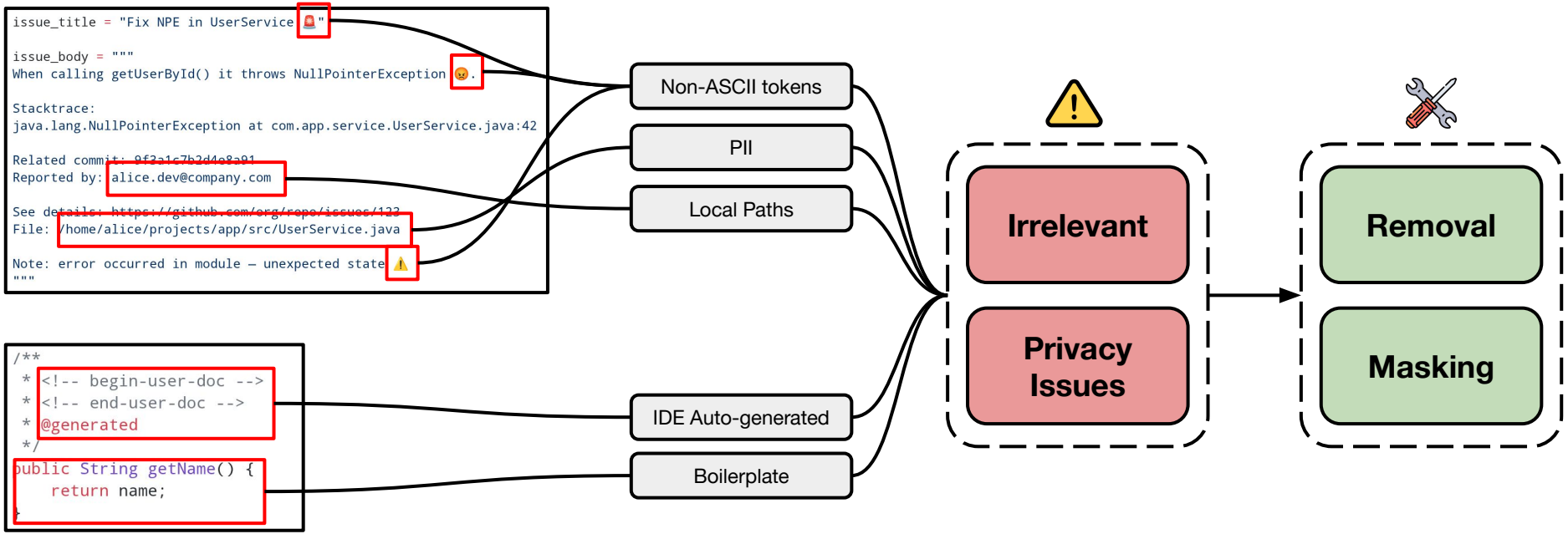
```
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public String getName() {
    return name;
}
```

- IDE Auto-generated
- Boilerplate

RQ1 Limited Informativeness

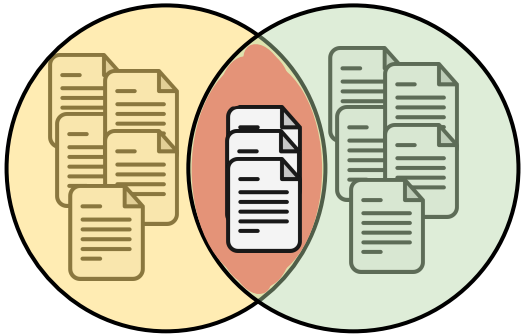


RQ1 Limited Informativeness



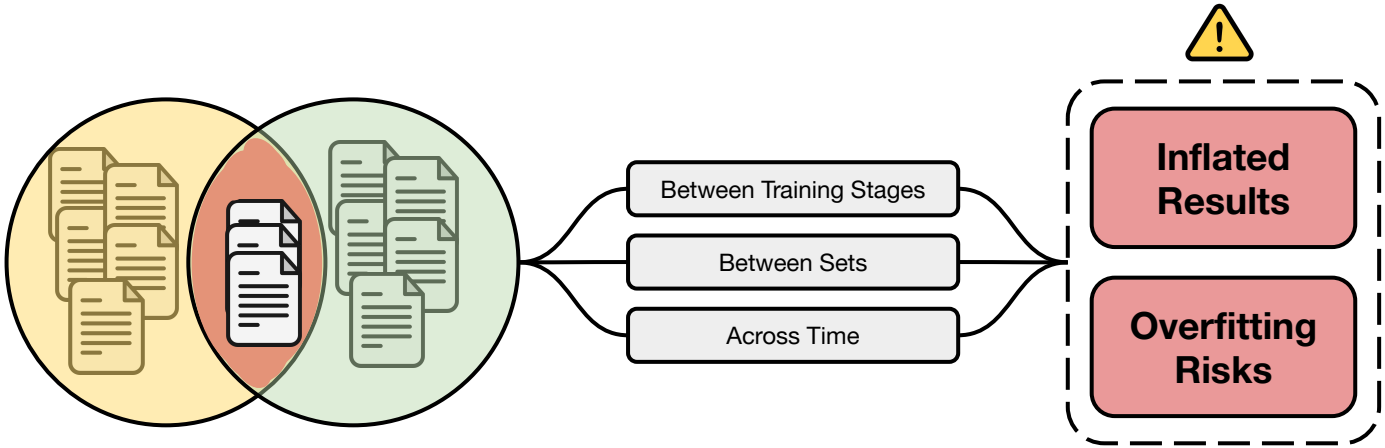
RQ1

Data Leakage



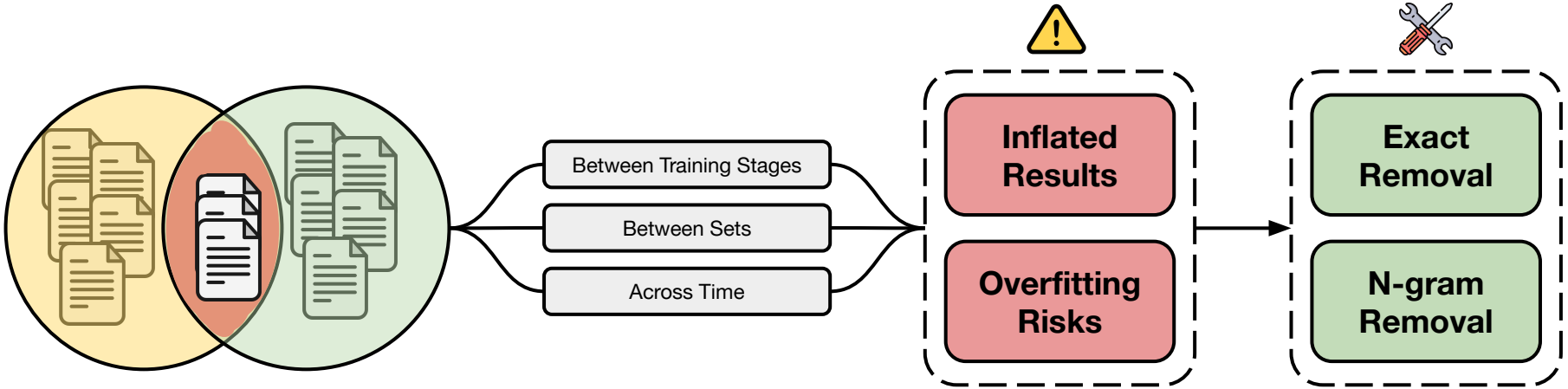
RQ1

Data Leakage



RQ1

Data Leakage



RQ1

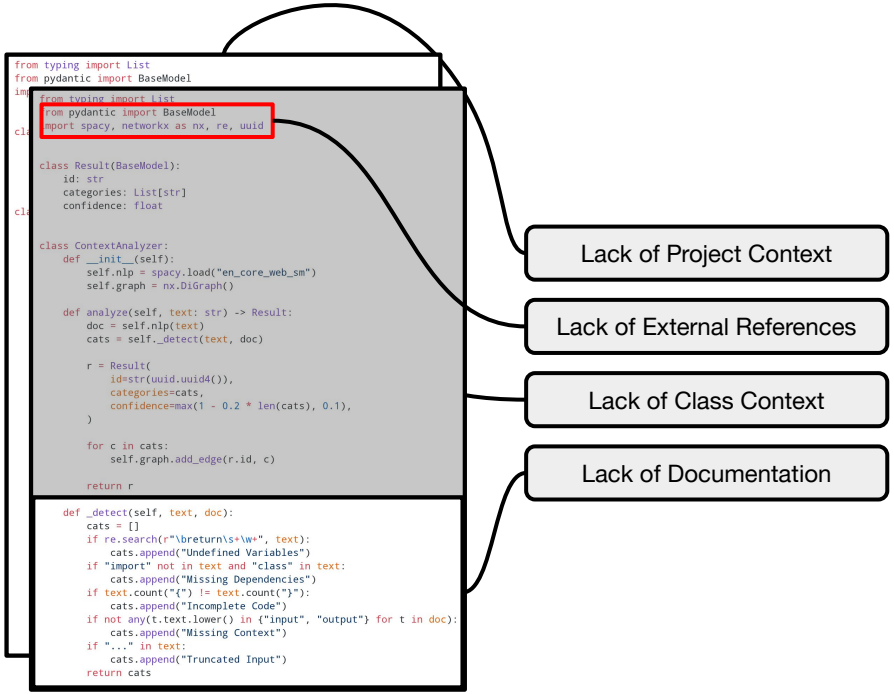
Lack of Context

```python
from typing import List
from pydantic import BaseModel
import spacy, networkx as nx, re, uuid


class Result(BaseModel):
    id: str
    categories: List[str]
    confidence: float


class ContextAnalyzer:
    def __init__(self):
        self.nlp = spacy.load("en_core_web_sm")
        self.graph = nx.DiGraph()

    def analyze(self, text: str) -> Result:
        doc = self.nlp(text)
        cats = self._detect(text, doc)

        r = Result(
            id=str(uuid.uuid4()),
            categories=cats,
            confidence=max(1 - 0.2 * len(cats), 0.1),
        )

        for c in cats:
            self.graph.add_edge(r.id, c)

        return r
```

```python
    def _detect(self, text, doc):
        cats = []
        if re.search(r"\breturn\s+\w+", text):
            cats.append("Undefined Variables")
        if "import" not in text and "class" in text:
            cats.append("Missing Dependencies")
        if text.count("{") != text.count("}"):
            cats.append("Incomplete Code")
        if not any(t.text.lower() in {"input", "output"} for t in doc):
            cats.append("Missing Context")
        if "..." in text:
            cats.append("Truncated Input")
        return cats
```

RQ1

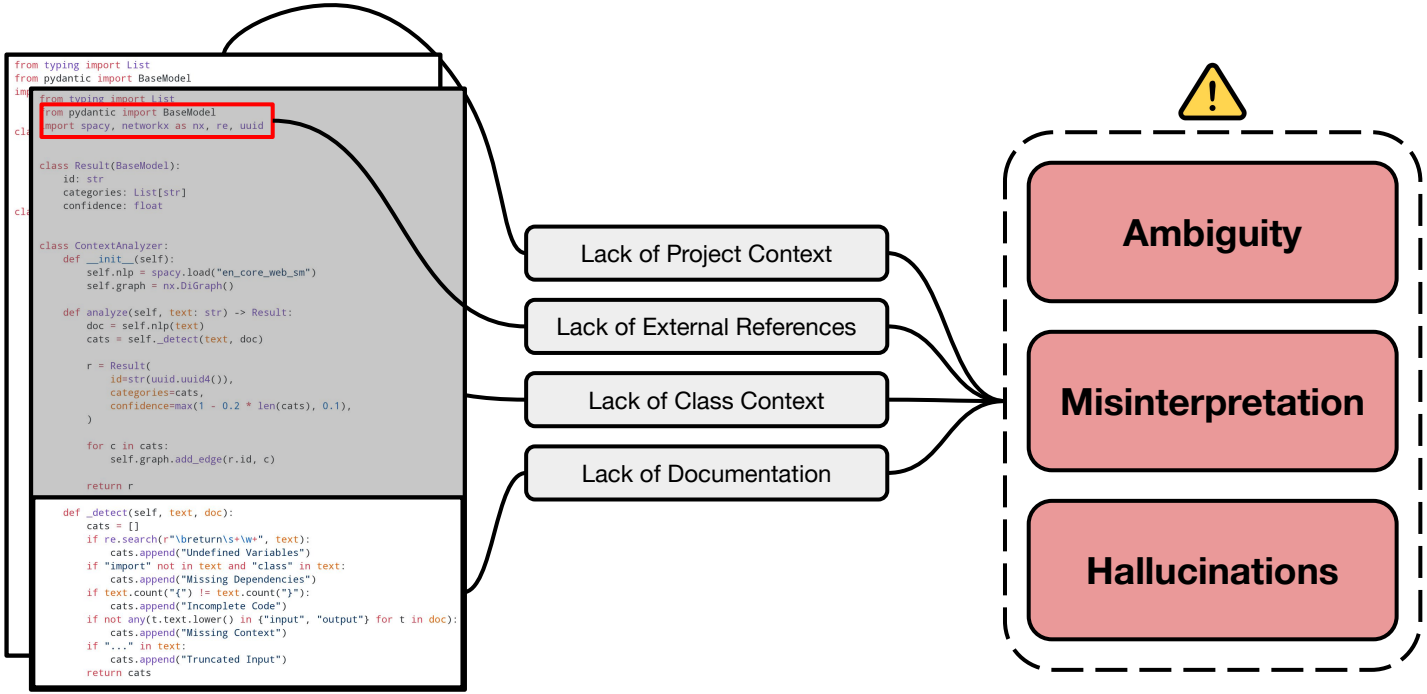
Lack of Context



- Lack of Project Context
- Lack of External References
- Lack of Class Context
- Lack of Documentation

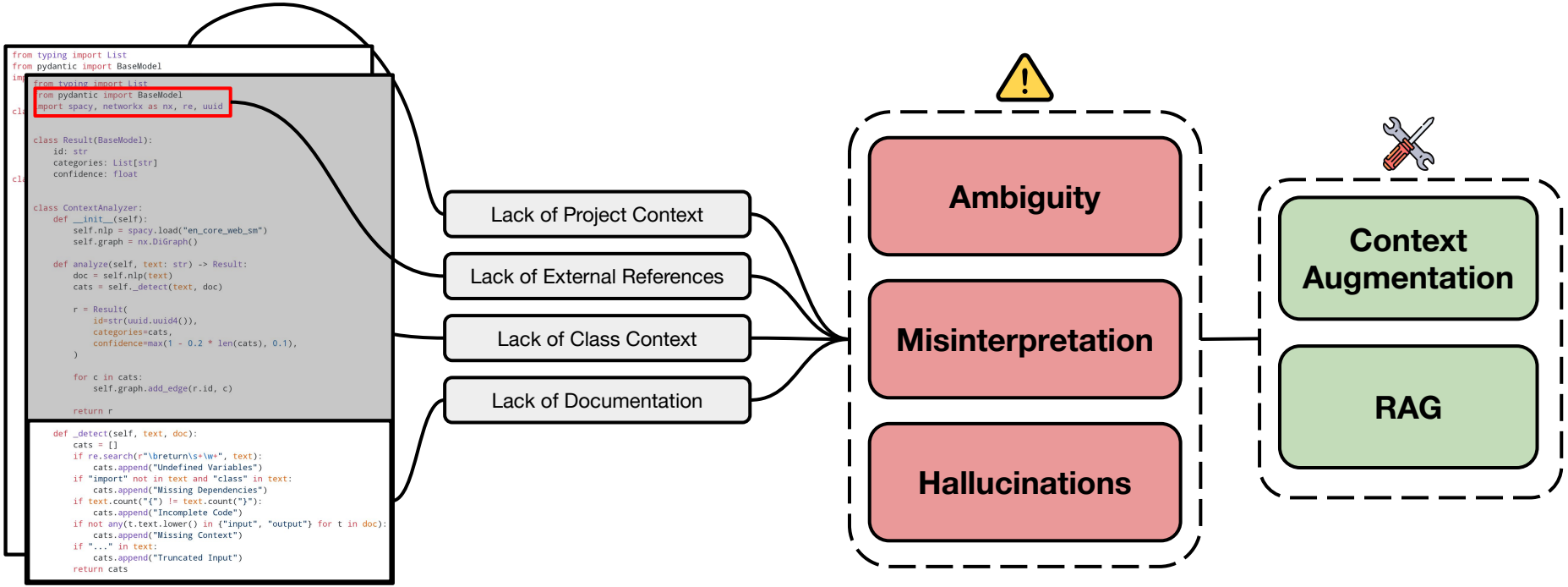
RQ1

Lack of Context



RQ1

Lack of Context



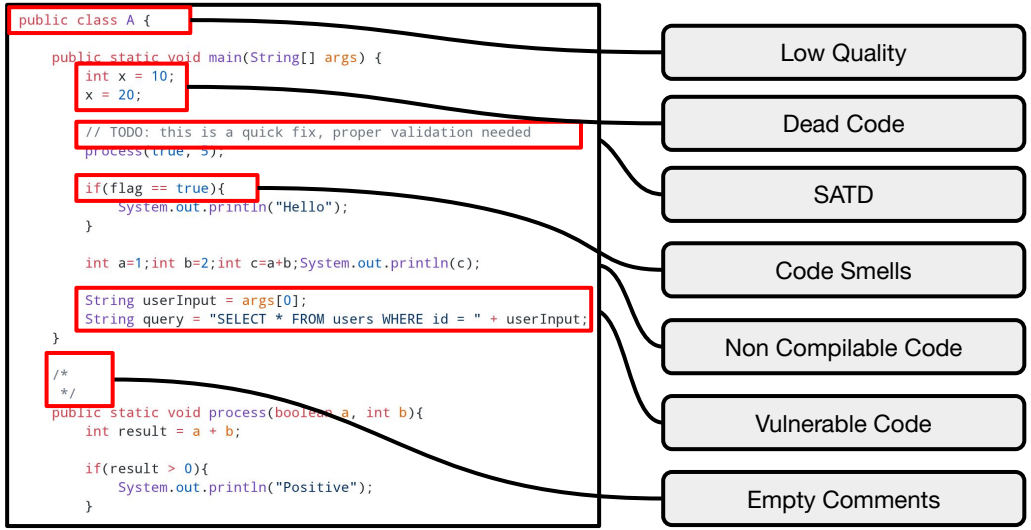
RQ1

Source Code Quality Issues

```java
public class A {

    public static void main(String[] args) {
        int x = 10;
        x = 20;

        // TODO: this is a quick fix, proper validation needed
        process(true, 5);

        if(flag == true){
            System.out.println("Hello");
        }

        int a=1;int b=2;int c=a+b;System.out.println(c);

        String userInput = args[0];
        String query = "SELECT * FROM users WHERE id = " + userInput;
    }

    /*
     */
    public static void process(boolean a, int b){
        int result = a + b;

        if(result > 0){
            System.out.println("Positive");
        }
```

RQ1

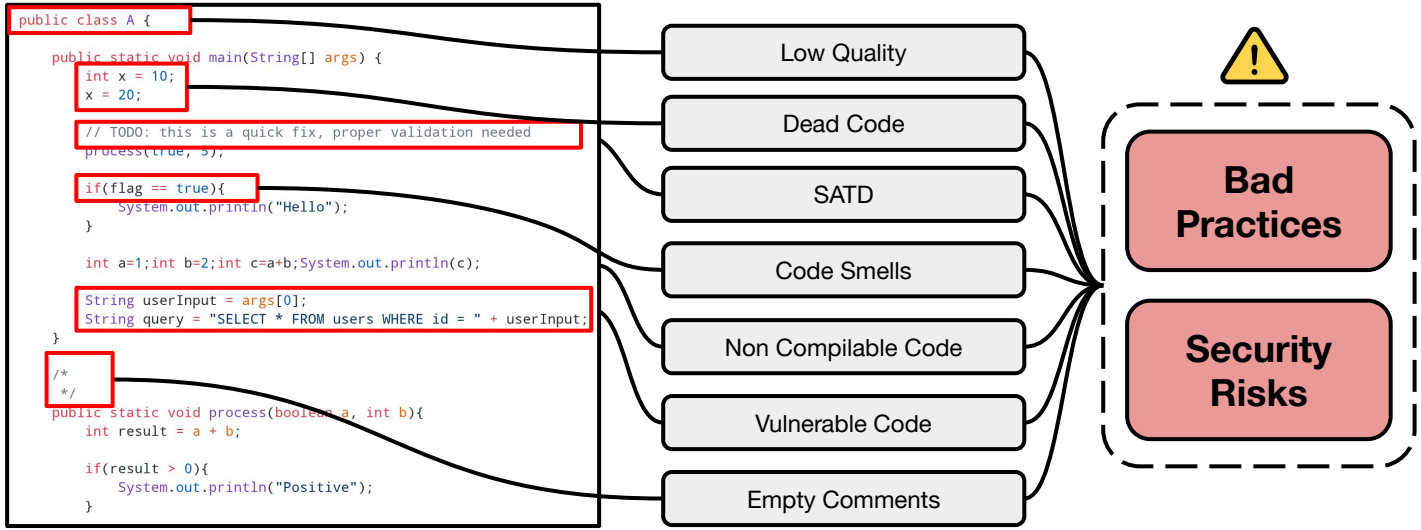
Source Code Quality Issues



- Low Quality
- Dead Code
- SATD
- Code Smells
- Non Compilable Code
- Vulnerable Code
- Empty Comments

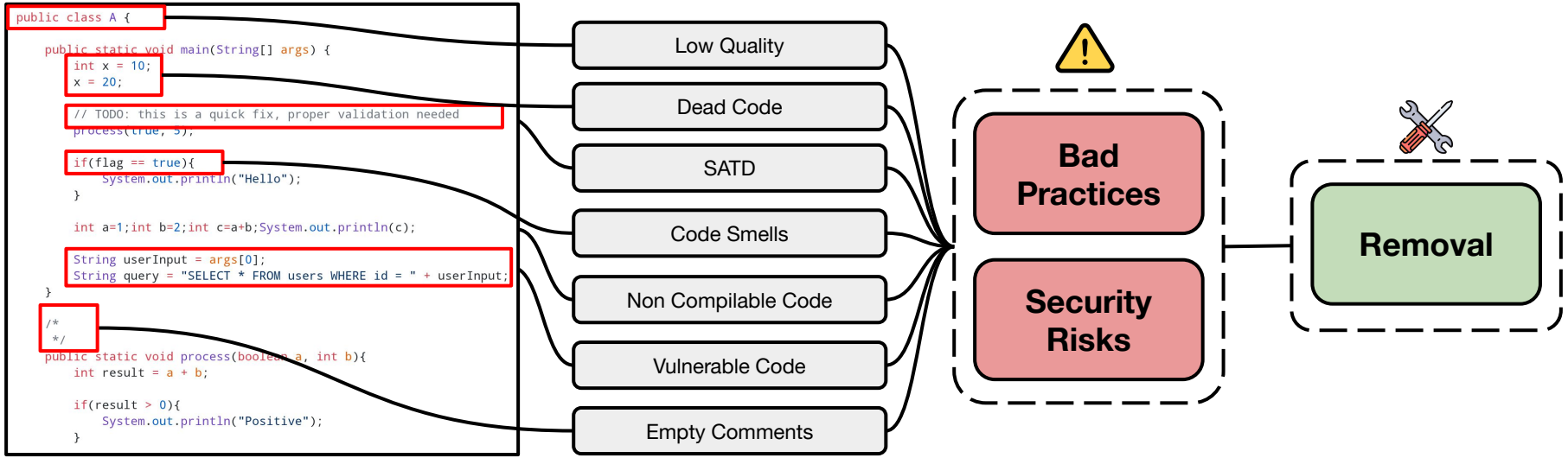
RQ1

Source Code Quality Issues



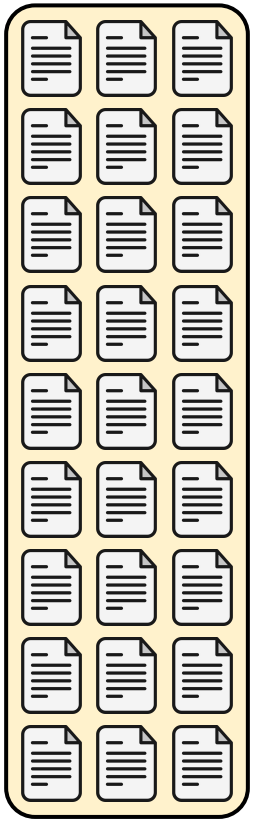
RQ1

Source Code Quality Issues



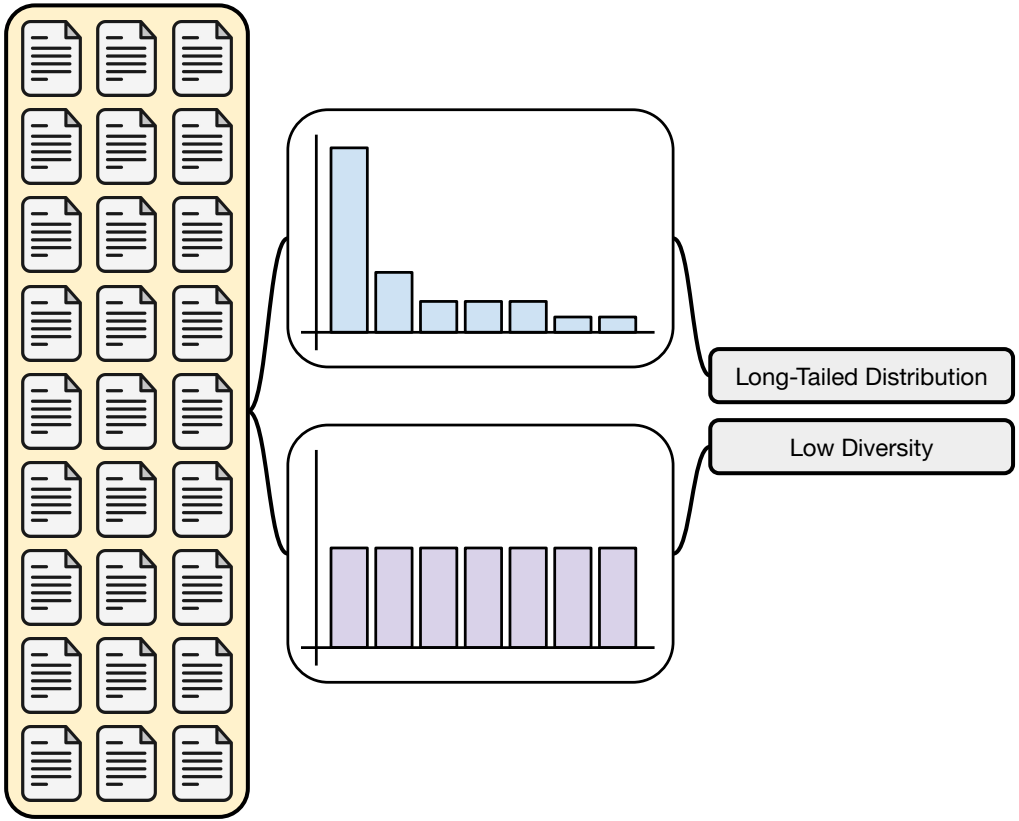
RQ1

Data Distribution Issues



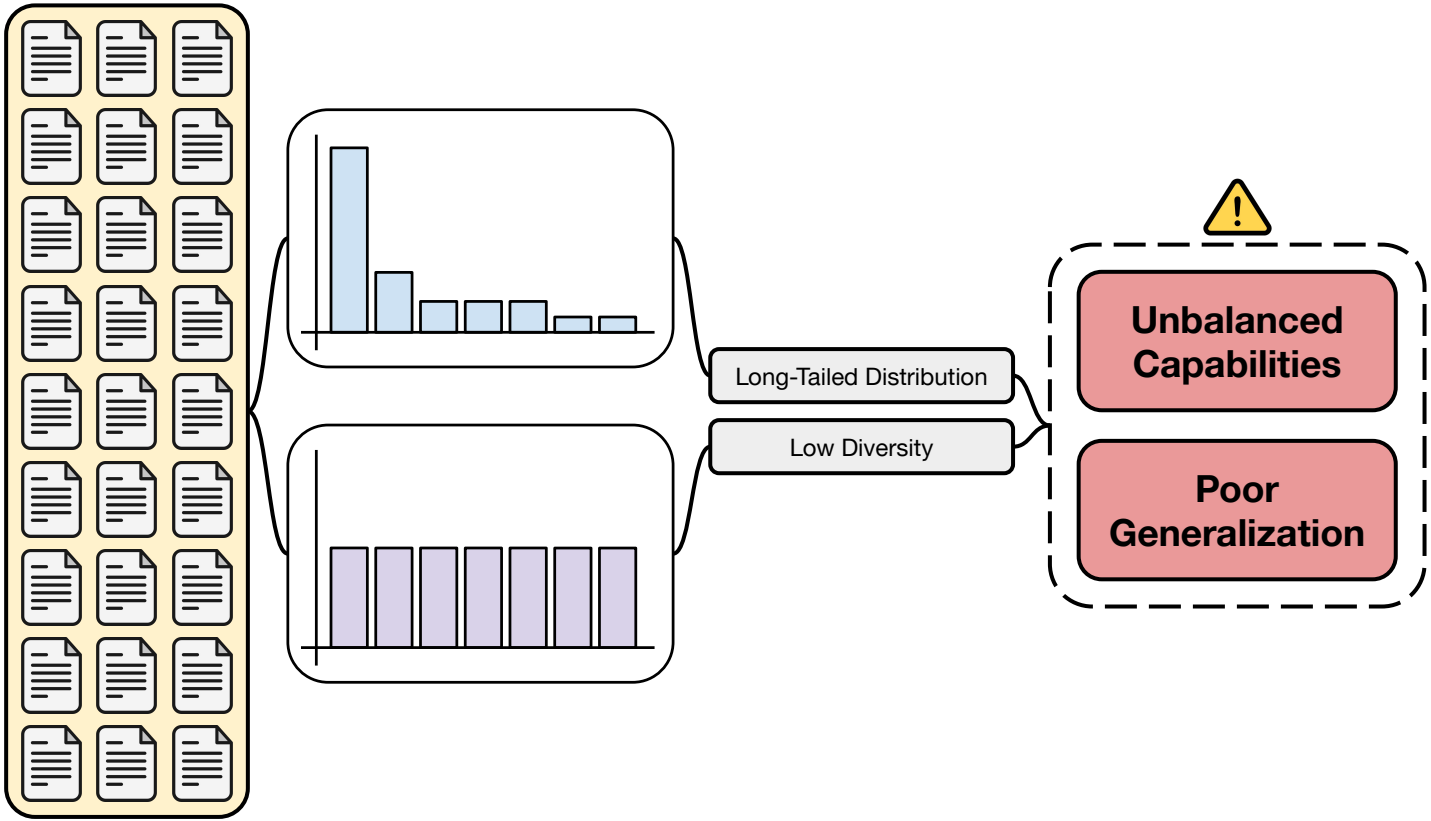
RQ1

Data Distribution Issues



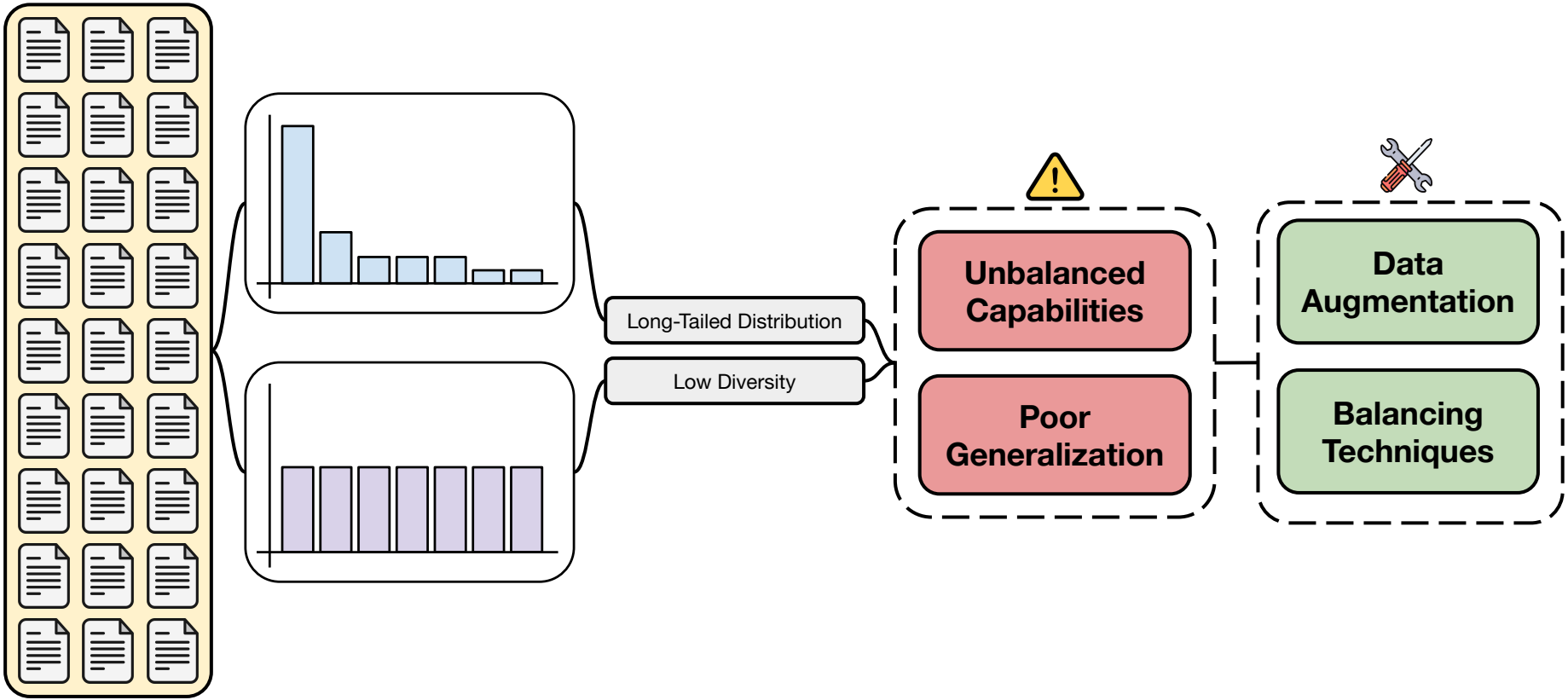
RQ1

Data Distribution Issues



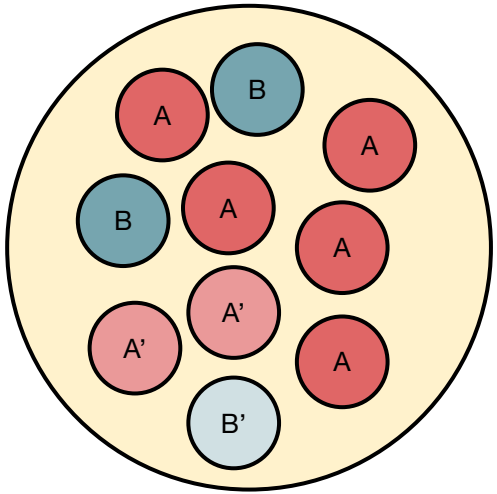
RQ1

Data Distribution Issues



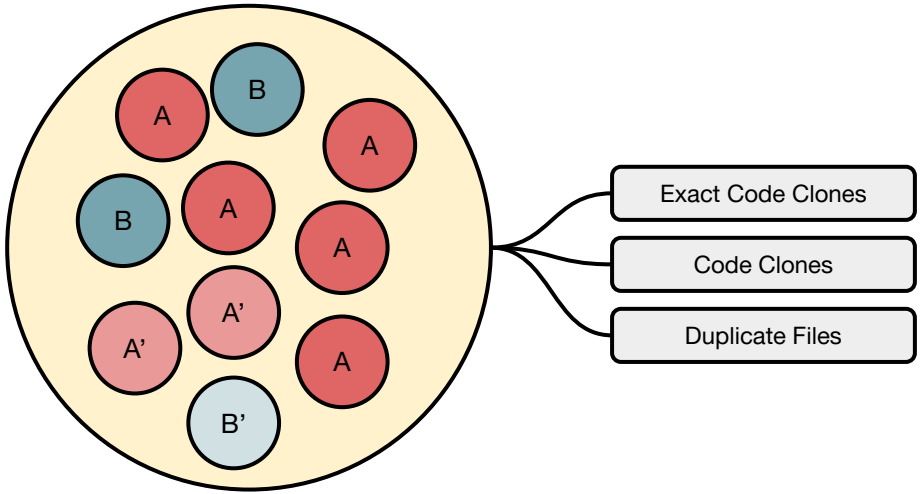
RQ1

(Near) Duplicated Instances



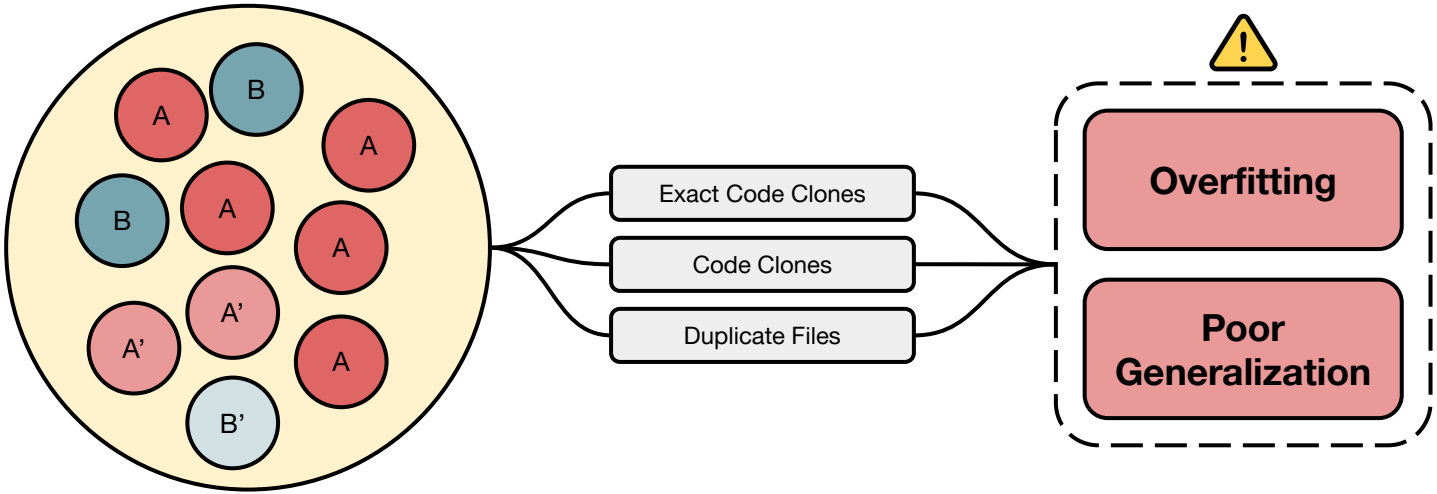
RQ1

(Near) Duplicated Instances



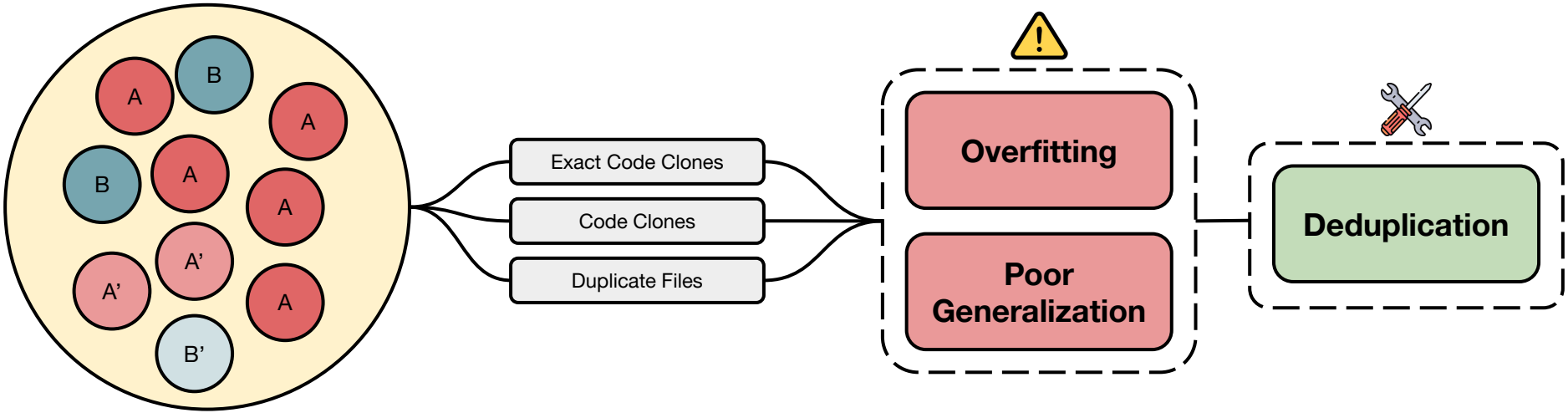
RQ1

(Near) Duplicated Instances



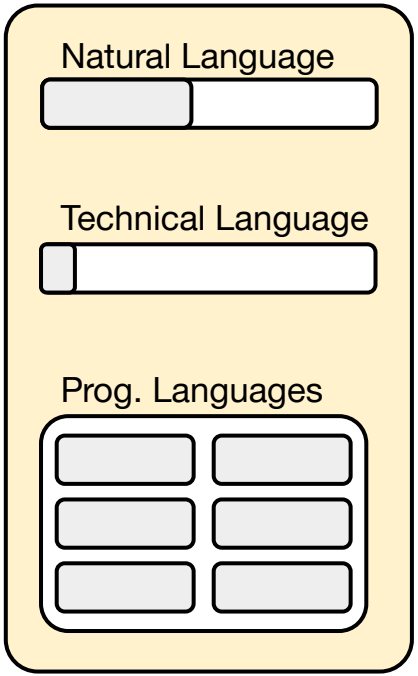
RQ1

(Near) Duplicated Instances



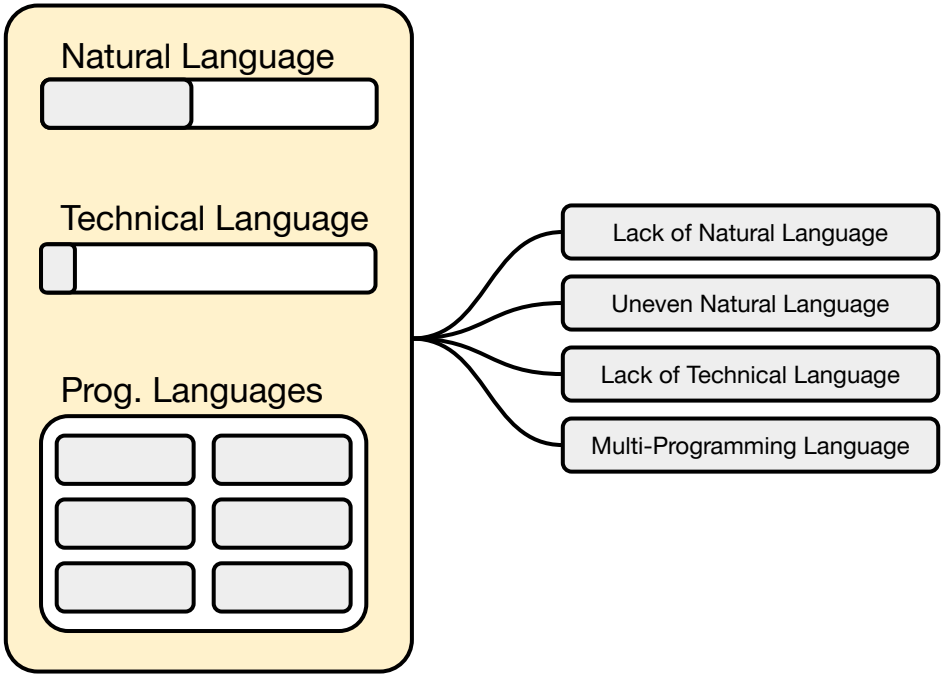
RQ1

Language Issues



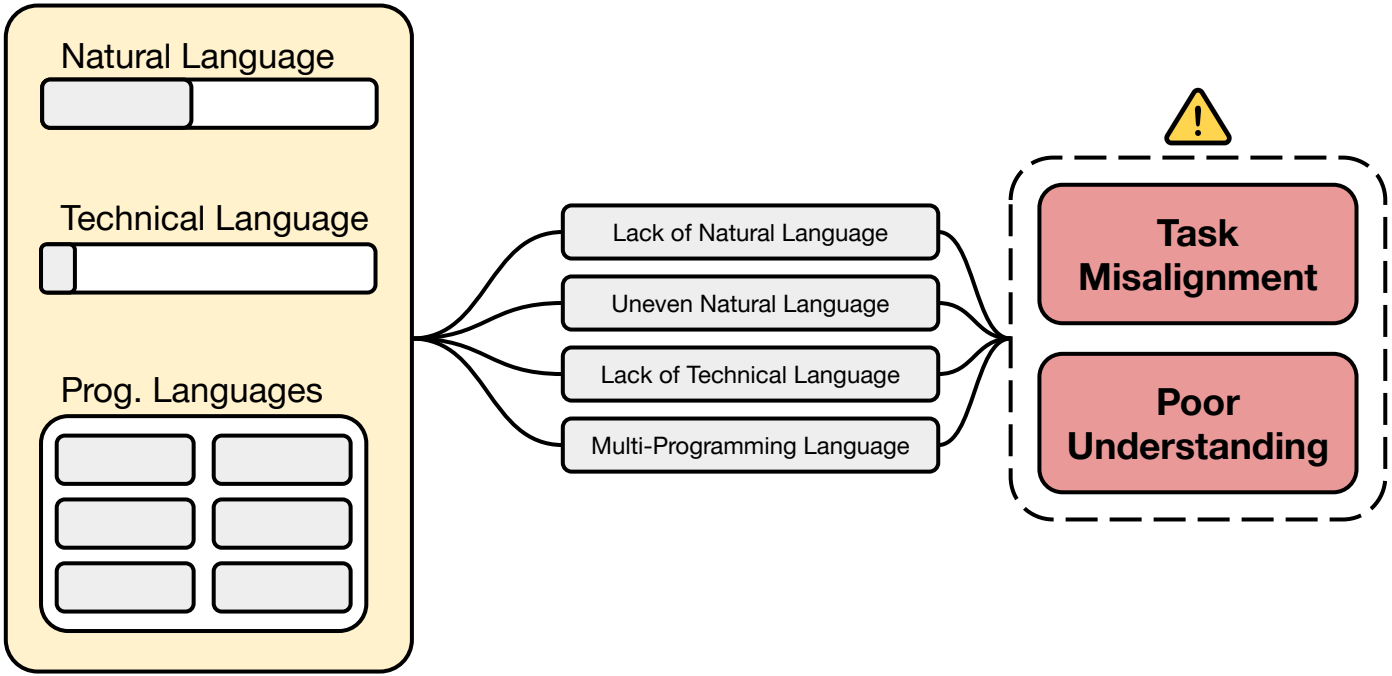
RQ1

Language Issues



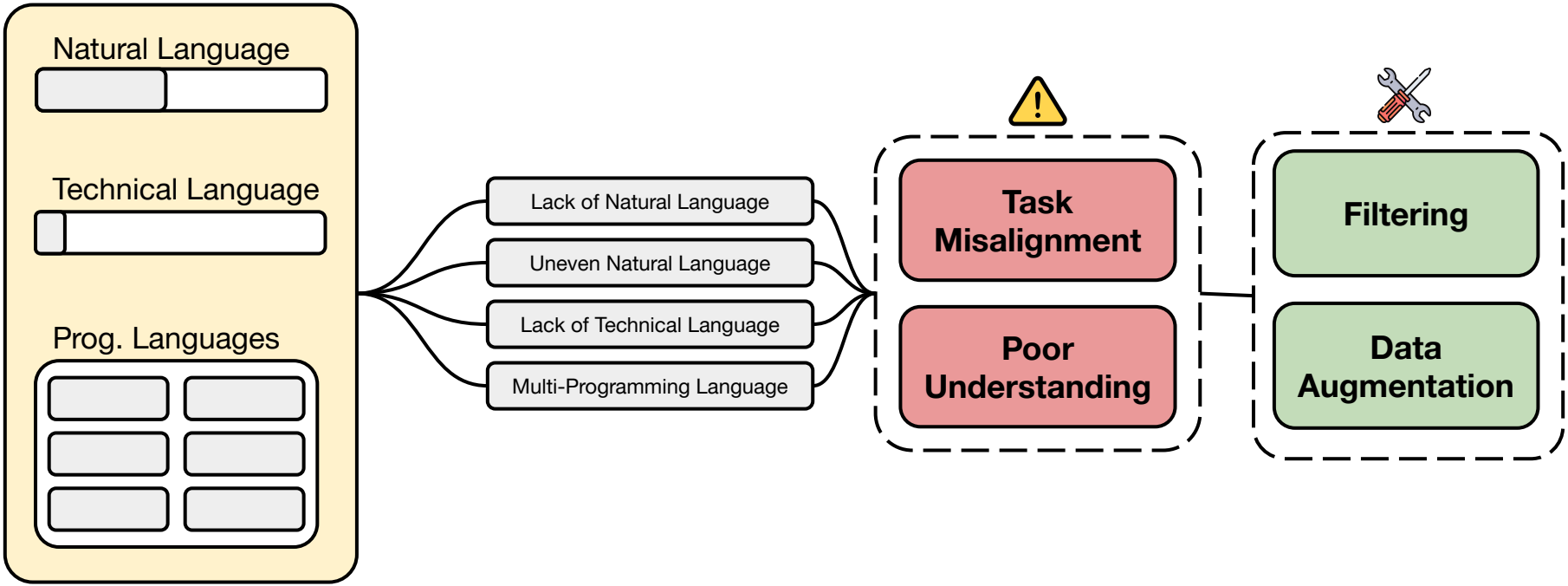
RQ1

Language Issues

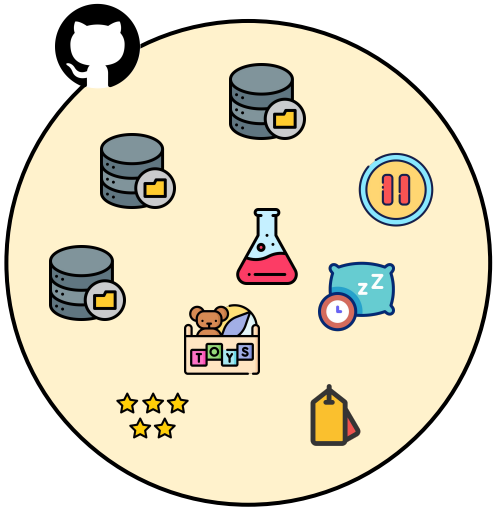


RQ1

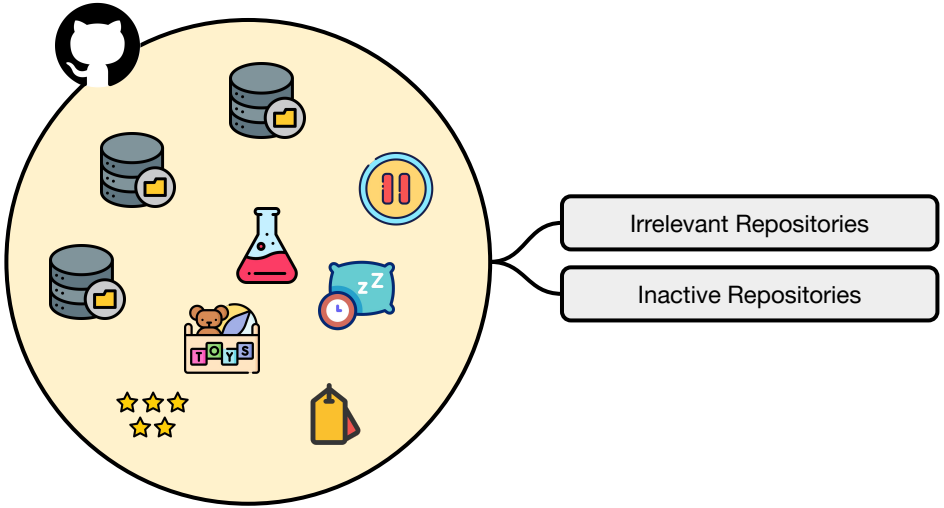
Language Issues



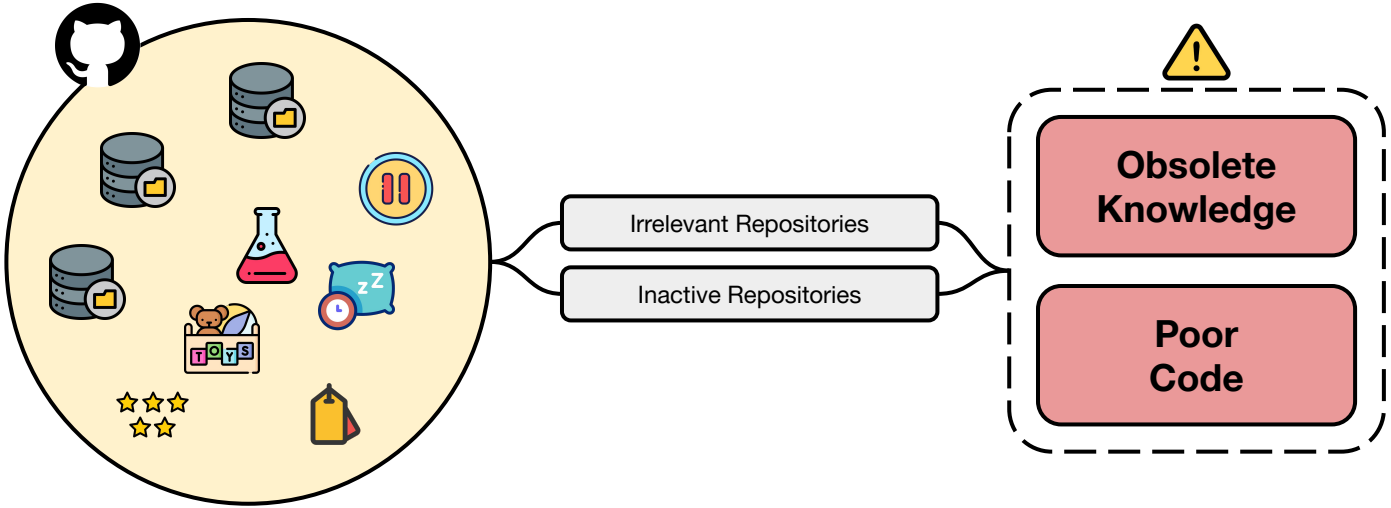
RQ1 Inadequate Source Repositories



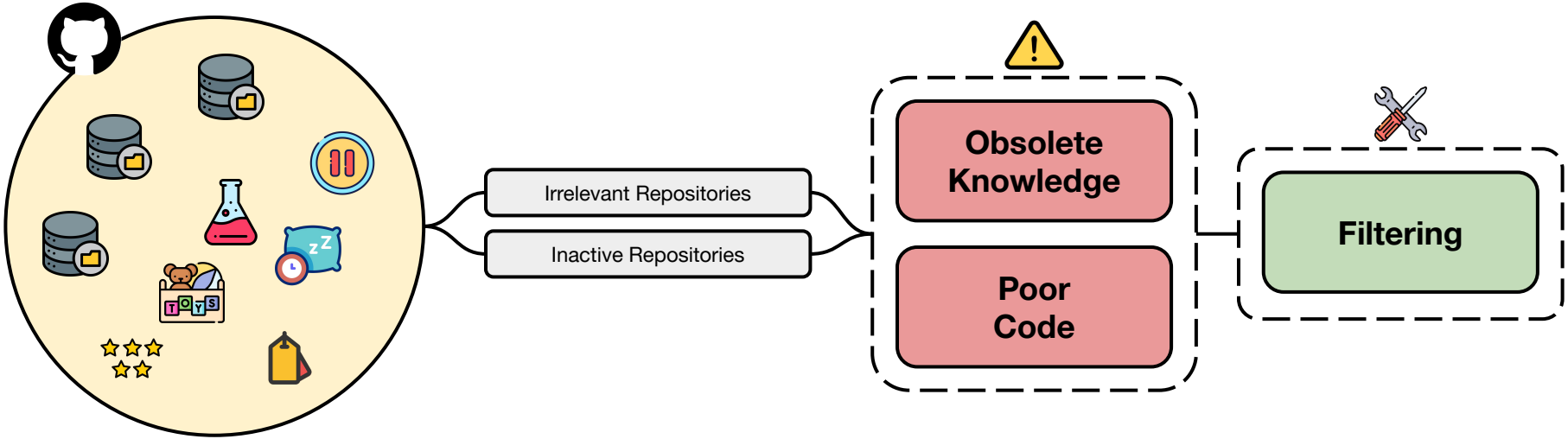
RQ1 Inadequate Source Repositories



RQ1 Inadequate Source Repositories

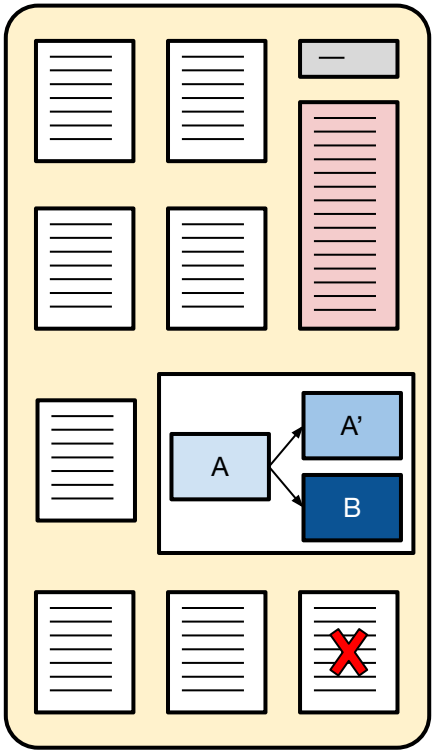


RQ1 Inadequate Source Repositories



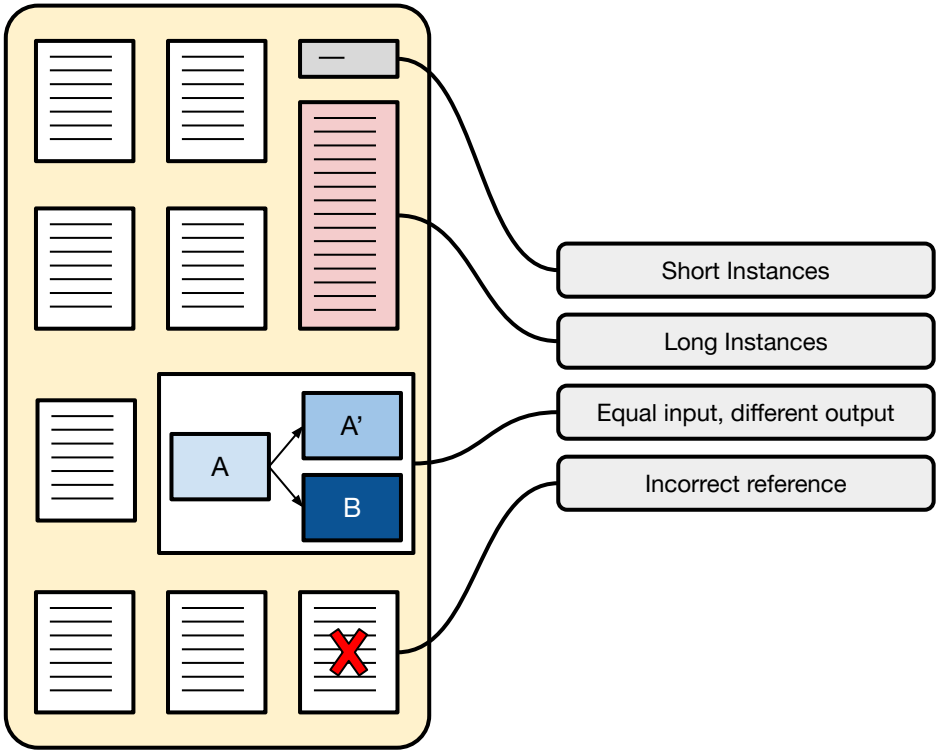
RQ1

Misleading Instances



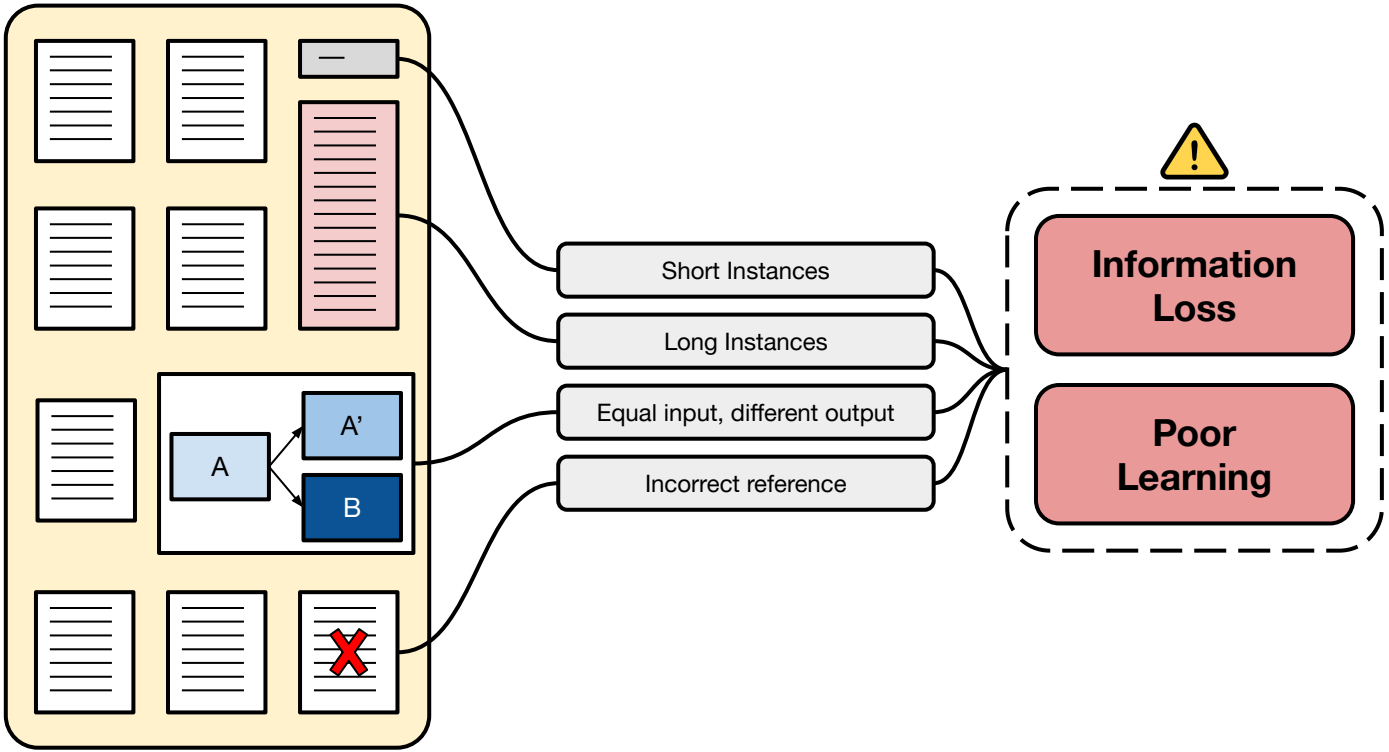
RQ1

Misleading Instances



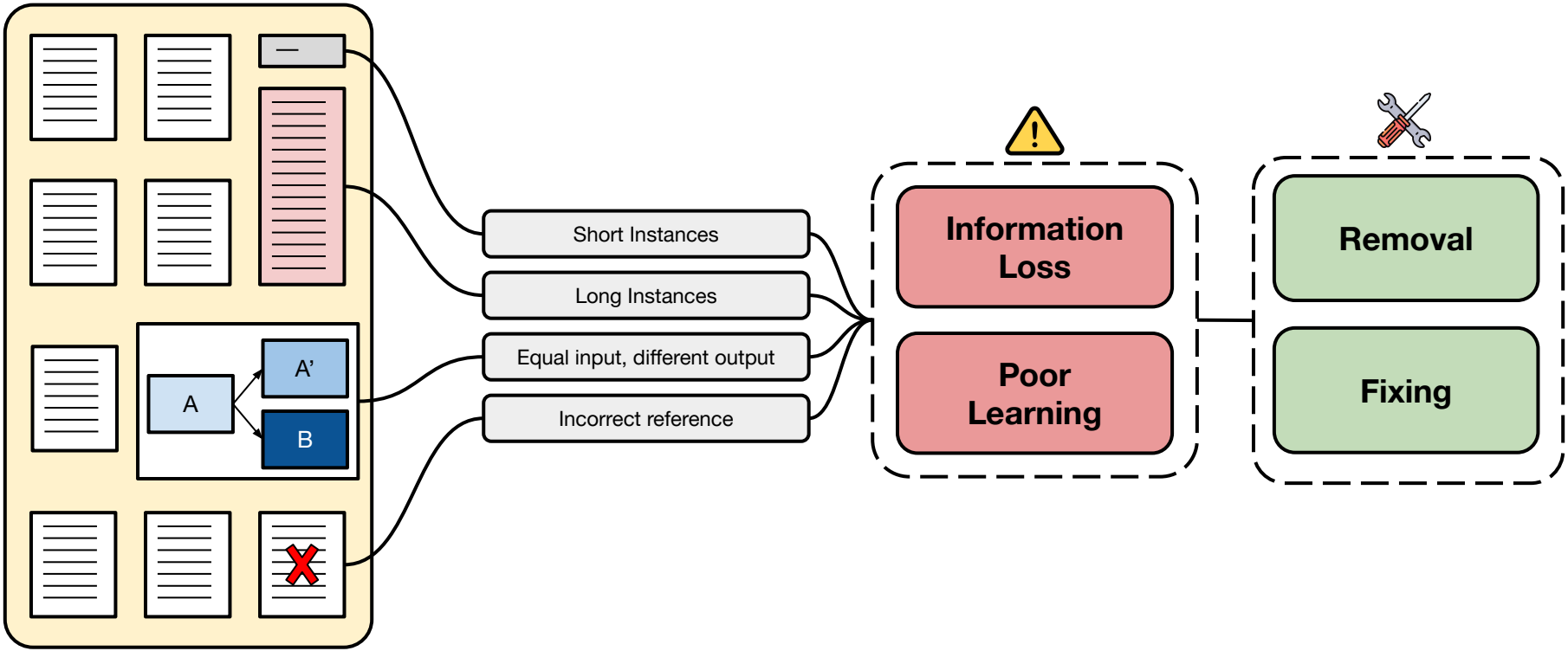
RQ1

Misleading Instances



RQ1

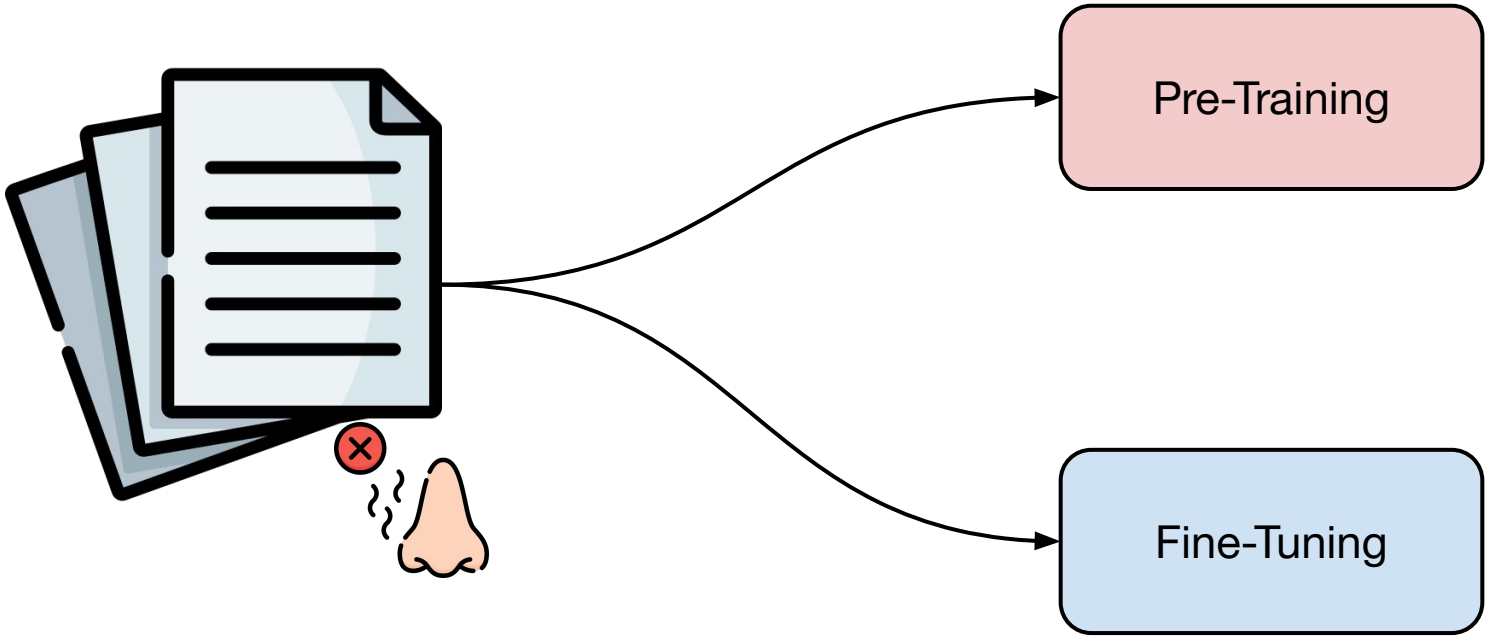
Misleading Instances



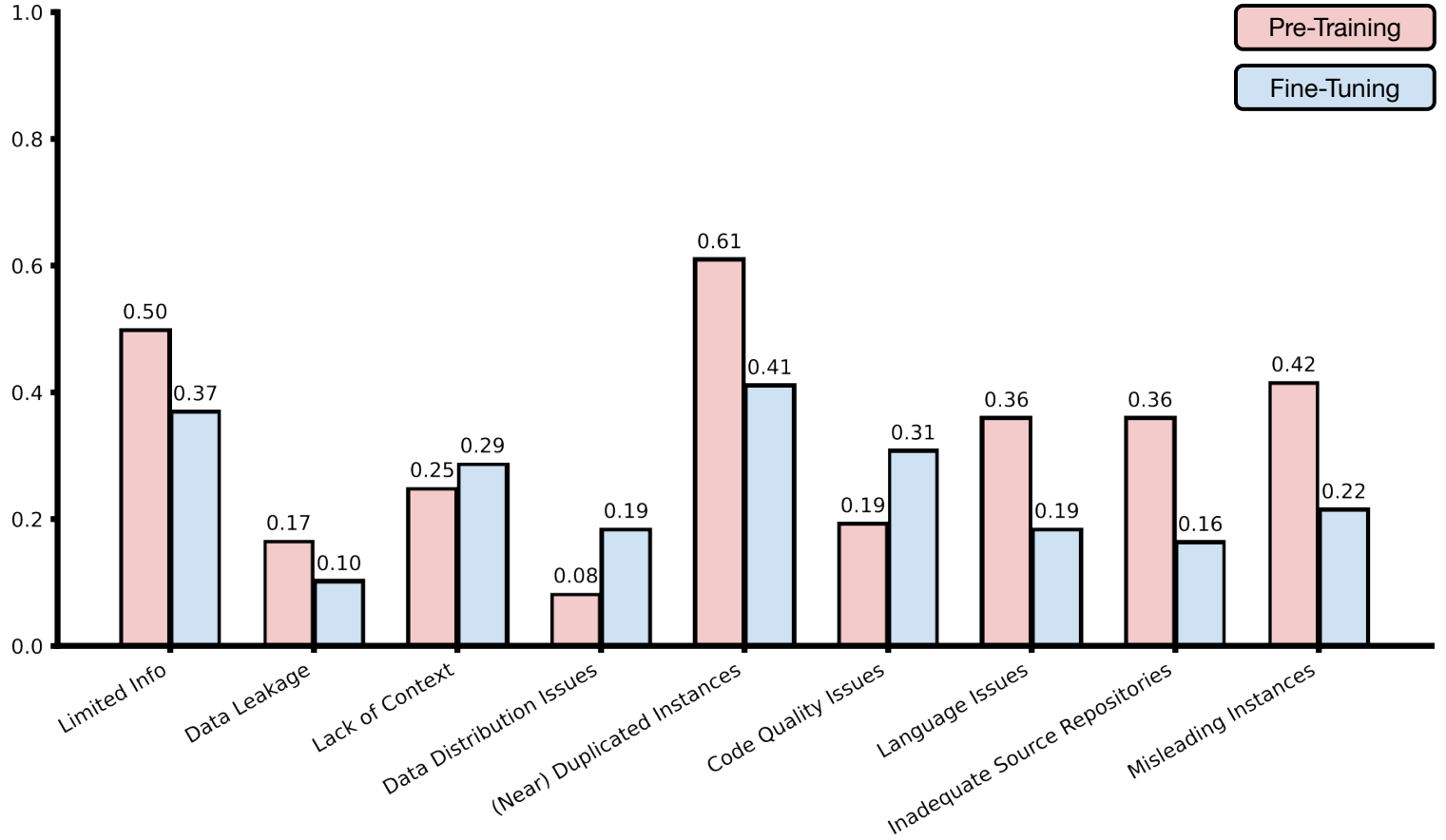
RQ2

How **frequent** is the use of strategies to **remove data smells** between **pre-training** and **fine-tuning**

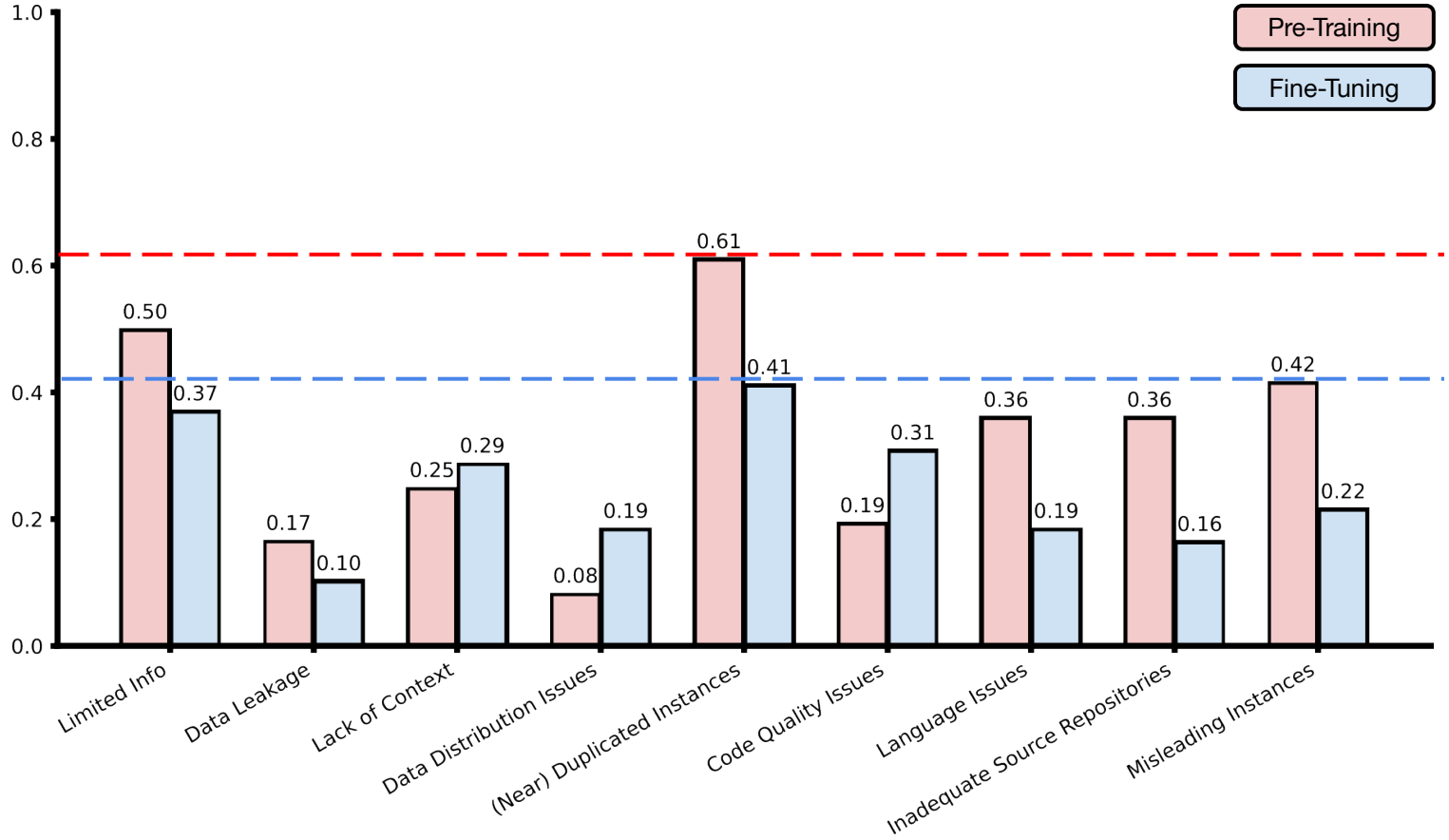
DESIGN



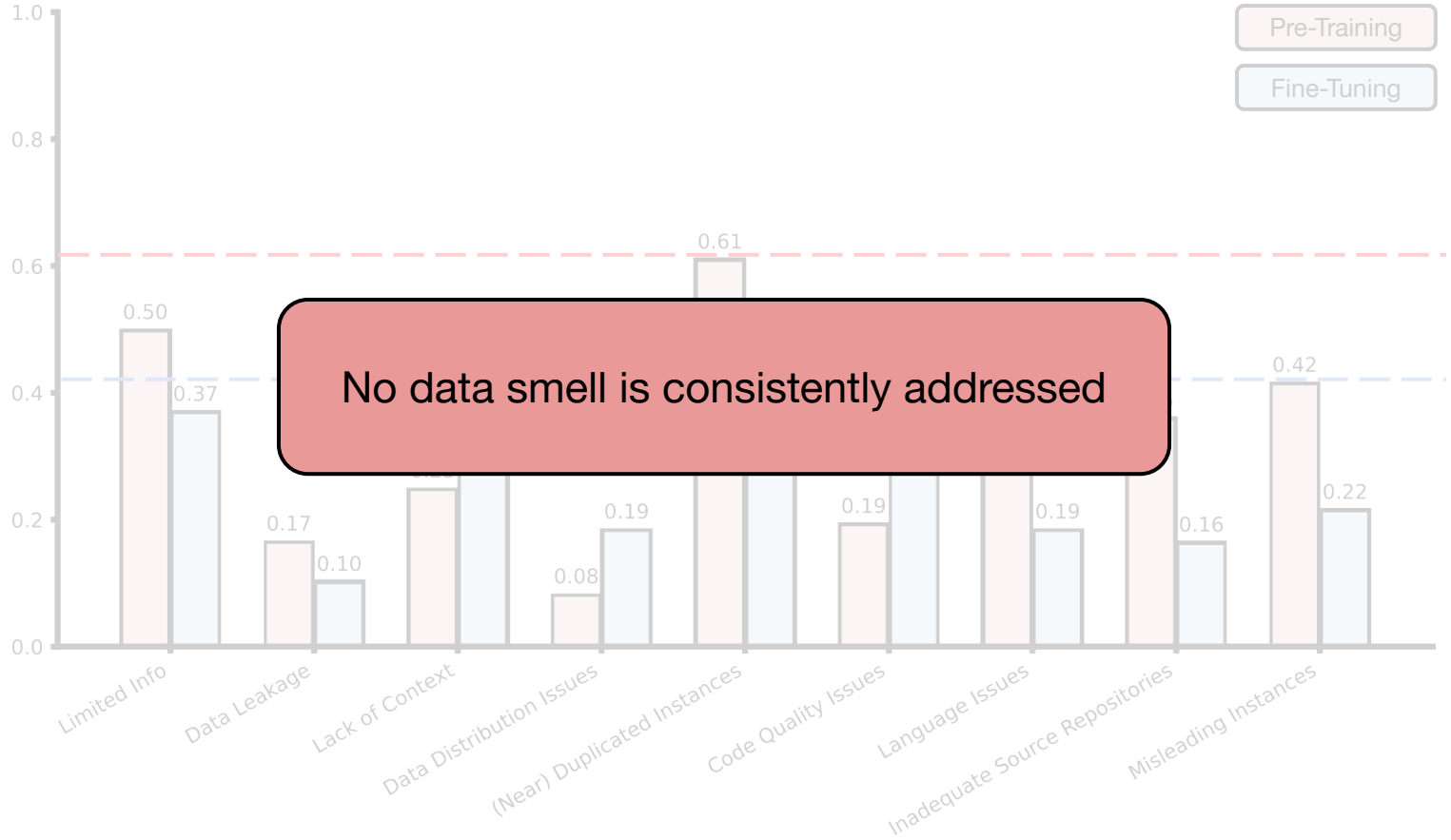
RQ2


Pre-Training
Fine-Tuning
1.0
0.8
0.6
0.4
0.2
0.0
0.50
0.37
0.17
0.10
0.25
0.29
0.08
0.19
0.61
0.41
0.19
0.31
0.36
0.19
0.36
0.16
0.42
0.22
Limited Info
Data Leakage
Lack of Context
Data Distribution Issues
(Near) Duplicated Instances
Code Quality Issues
Language Issues
Inadequate Source Repositories
Misleading Instances

RQ2


Pre-Training
Fine-Tuning
1.0
0.8
0.6
0.4
0.2
0.0
0.50
0.37
0.17
0.10
0.25
0.29
0.08
0.19
0.61
0.41
0.19
0.31
0.36
0.19
0.36
0.16
0.42
0.22
Limited Info
Data Leakage
Lack of Context
Data Distribution Issues
(Near) Duplicated Instances
Code Quality Issues
Language Issues
Inadequate Source Repositories
Misleading Instances

RQ2

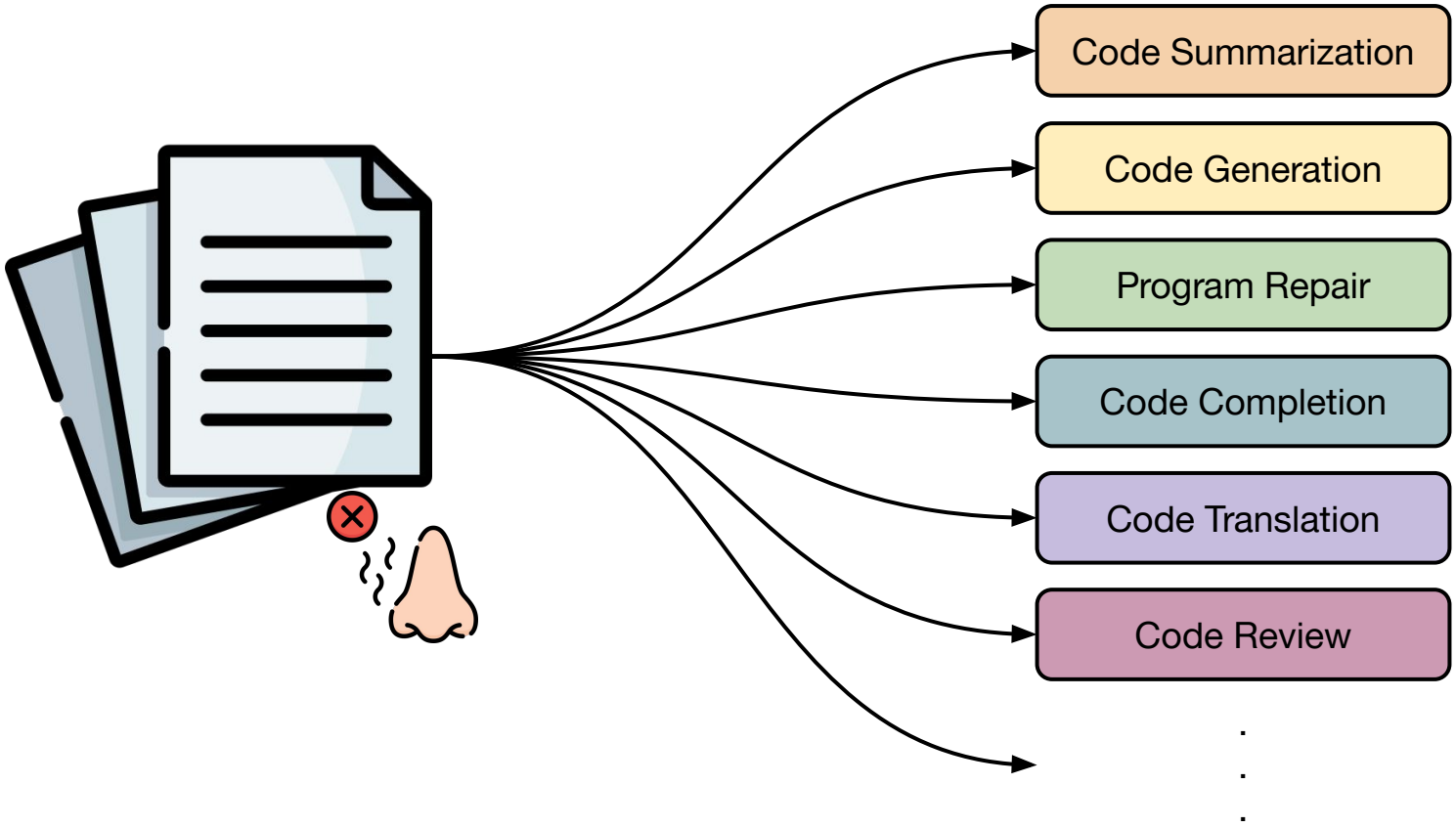


No data smell is consistently addressed

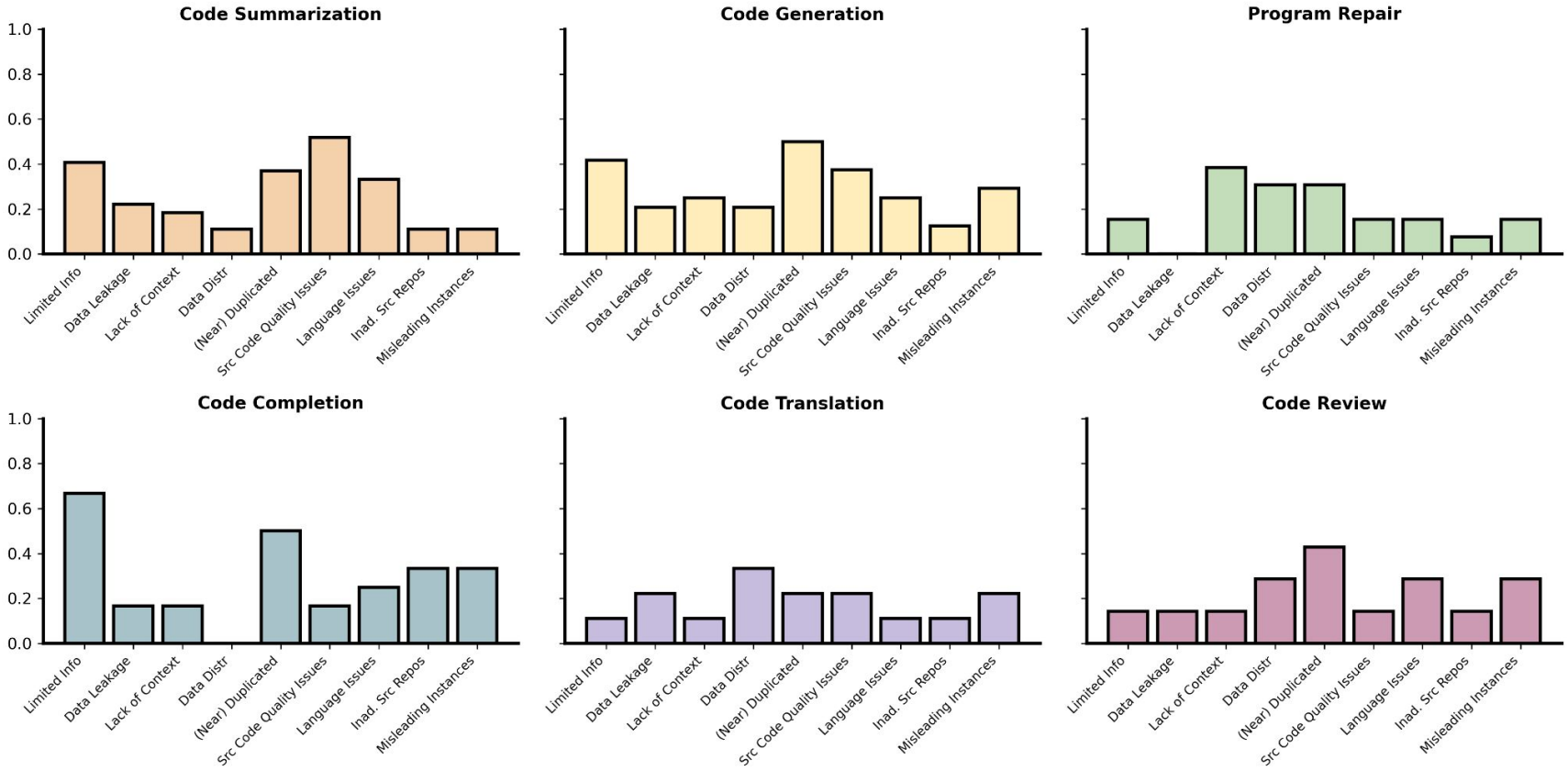
RQ3

How **frequent** is the use of strategies **to remove data smells** among different **coding tasks**

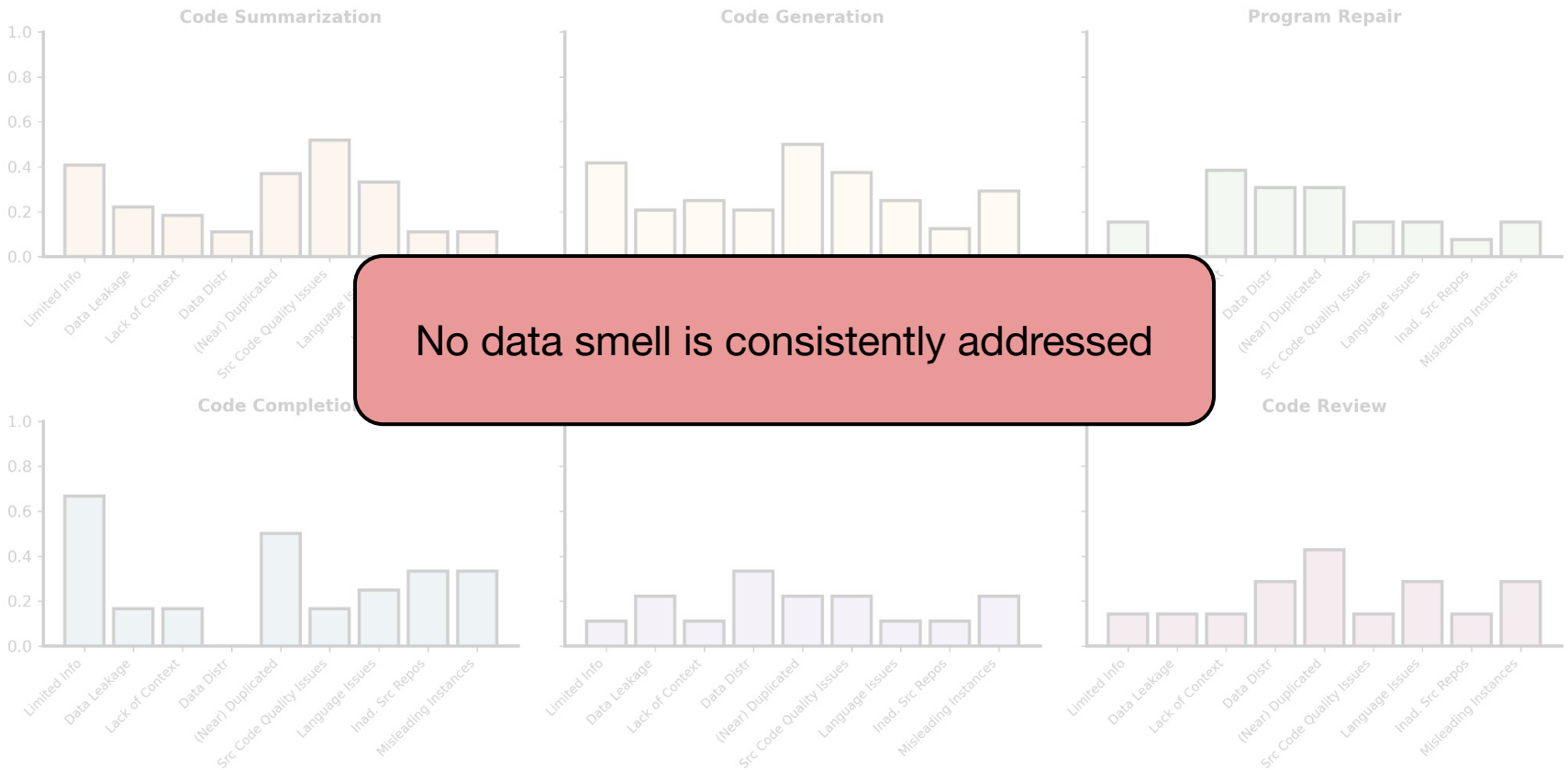
DESIGN


Code Summarization
Code Generation
Program Repair
Code Completion
Code Translation
Code Review
.
.
.

RQ3


Code Summarization
Code Generation
Program Repair
Code Completion
Code Translation
Code Review
1.0
0.8
0.6
0.4
0.2
0.0
Limited Info
Data Leakage
Lack of Context
Data Distr
(Near) Duplicated
Src Code Quality Issues
Language Issues
Inad. Src Repos
Misleading Instances

RQ3



No data smell is consistently addressed

OUTCOMES

Discrepancy:

A discrepancy is found between the data smells addressed during **pre-training** and **fine-tuning** stages.

It is evidenced that many **code-related tasks DO not** take into account many data smells.

Shallowness:

Many cleaning techniques are **shallow** and, consequently, no precise.

Beyond Effectiveness:

It is needed to measure the impact of data smells also on **other aspects** (*i.e.*, security, privacy, robustness).

Quality Improvement:

To improve data quality, a canonical approach is to **remove low-quality instances**.

On the other hand, **augment instances missing features** can be crucial.

OUTCOMES

Discrepancy:

A discrepancy is found between the data smells addressed during **pre-training** and **fine-tuning** stages.

It is evidenced that many **code-related tasks DO not** take into account many data smells.

Shallowness:

Many cleaning techniques are **shallow** and, consequently, no precise.

Beyond Effectiveness:

It is needed to measure the impact of data smells also on **other aspects** (*i.e.*, security, privacy, robustness).

Quality Improvement:

To improve data quality, a canonical approach is to **remove low-quality instances**.

On the other hand, **augment instances missing features** can be crucial.

OUTCOMES

Discrepancy:

A discrepancy is found between the data smells addressed during **pre-training** and **fine-tuning** stages.

It is evidenced that many **code-related tasks DO not** take into account many data smells.

Shallowness:

Many cleaning techniques are **shallow** and, consequently, no precise.

Beyond Effectiveness:

It is needed to measure the impact of data smells also on **other aspects** (*i.e.*, security, privacy, robustness).

Quality Improvement:

To improve data quality, a canonical approach is to **remove low-quality instances**.

On the other hand, **augment instances missing features** can be crucial.

OUTCOMES

Discrepancy:

A discrepancy is found between the data smells addressed during **pre-training** and **fine-tuning** stages.

It is evidenced that many **code-related tasks DO not** take into account many data smells.

Shallowness:

Many cleaning techniques are **shallow** and, consequently, no precise.

Beyond Effectiveness:

It is needed to measure the impact of data smells also on **other aspects** (*i.e.*, security, privacy, robustness).

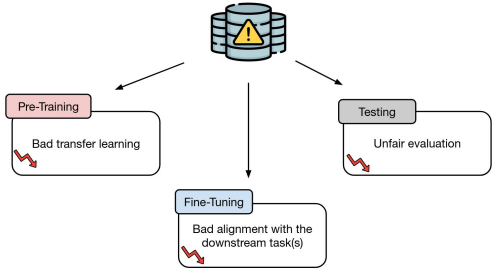
Quality Improvement:

To improve data quality, a canonical approach is to **remove low-quality instances**.

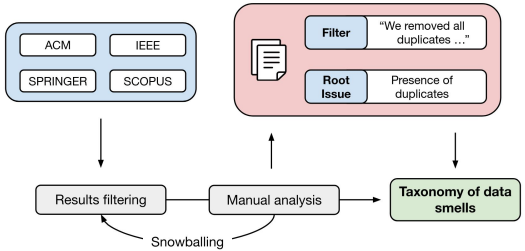
On the other hand, **augment instances missing features** can be crucial.

SUMMARY

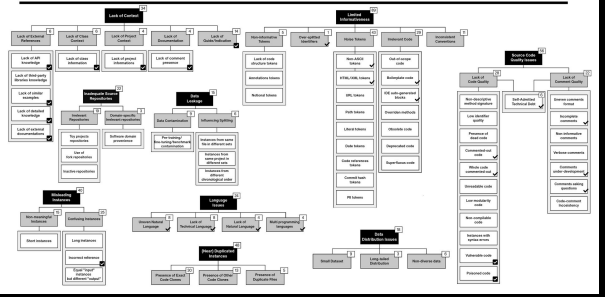
DATA IMPORTANCE



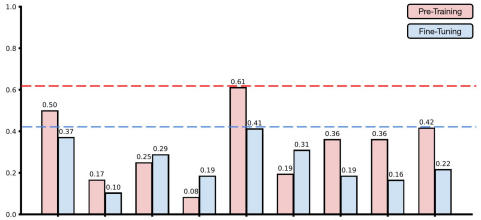
DESIGN



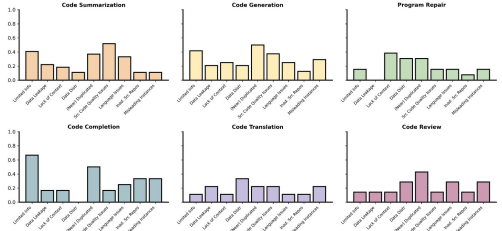
RQ1



RQ2



RQ3



OUTCOMES

Discrepancy:

A discrepancy is found between the data smells addressed during **pre-training** and **fine-tuning** stages.

It is evidenced that many **code-related tasks DO not** take into account many data smells.

Shallowness:

Many cleaning techniques are **shallow** and, consequently, no precise.

Beyond Effectiveness:

It is needed to measure the impact of data smells also on **other aspects** (*i.e.*, security, privacy, robustness).

Quality Improvement:

To improve data quality, a canonical approach is to **remove low-quality instances**.

On the other hand, **augment instances missing features** can be crucial.