Intelligent Run-time Reliability Engineering for Python Software

PENG, Yun

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
July 2024

Thesis Assessment Committee

Professor XU Qiang (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor MENG Wei (Committee Member)

Professor ZHANG Lingming (External Examiner)

Abstract of thesis entitled:

Intelligent Run-time Reliability Engineering for Python Software

Submitted by PENG, Yun

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in July 2024

Python has emerged as a widely favored dynamic programming language, as demonstrated by its rank as the second most utilized language on GitHub since 2019. Developers working with Python derive substantial advantages from its dynamic features. Among these, the dynamic type system and the dynamic run-time environment stand out as particularly significant. The dynamic type system eliminates the need for explicit type declarations, thereby simplifying the development of generic functions. The dynamic run-time environment removes the necessity for compilation during the development phase, facilitating the integration of the latest third-party packages. These features collectively enhance fast prototyping capabilities, reducing the effort required to develop Python software. This streamlined development process significantly contributes to Python's popularity in software engineering.

However, the flexibility afforded by Python's dynamic features is not without drawbacks, as it may compromise the reliability of the software. The absence of strict requirements like type declarations and compilation increases the susceptibility of Python software to reliability issues such as type errors and run-time environment conflicts. In this thesis, we propose methods to detect and mitigate the reliability issues arising from the dynamic type system and the dynamic

runtime environment inherent to Python. Our approach aims to bolster the reliability of Python software, addressing key vulnerabilities introduced by its dynamic nature.

Firstly, we introduce a hybrid type inference method termed HiTyper, which synergizes static type inference with neural predictions. Type inference statically assigns types to variables in the code, enabling the detection of type errors by checking potential type conflicts via type checking tools. Our approach merges the precision of static type inference with the extensive coverage and effectiveness of neural predictions. To facilitate this integration, we document type dependencies among variables within each function and represent these relationships through Type Dependency Graphs (TDGs). Utilizing TDGs allows for the seamless incorporation of type inference rules at the nodes to perform static inference, alongside type rejection rules to filter out incorrect neural predictions. HiTyper iteratively executes static inference and neural prediction until the TDG is fully resolved, thereby determining the types for all variables in the code. This method enhances the type reliability of Python software by preventing potential type errors.

Secondly, we propose a generative type inference approach named TypeGen to further improve the performance of HiTyper. Recognizing that the performance upper bound of HiTyper is contingent upon the performance of the underlying deep learning models, we design a more advanced generative approach based on powerful large language models. This approach generates chain-of-thought (COT) prompts by converting the procedural steps of static type analysis into structured prompts derived from Type Dependency Graphs (TDGs). This format allows the language models to internalize the methodologies employed by static analysis for type inference. TypeGen enhances this learning process by incorporating code snippets and type hints alongside the COT prompts, creating

comprehensive example prompts based on human annotations. These example prompts facilitate in-context learning, enabling the language models to generate similar COT prompts that include the final type predictions. This innovative approach not only improves the performance of neural type predictions but also expands the capabilities of HiTyper in handling complex type inference scenarios.

Thirdly, we propose a domain-aware prompt-based program repair method named TypeFix for repairing existing Python type errors. Prompt-based program repair techniques have demonstrated effectiveness, however, their performance largely hinges on the quality of the prompt templates used for inserting masks. TypeFix enhances the prompt-based approach by first employing a novel hierarchical clustering algorithm to mine generalized fix templates. These templates capture common editing patterns and contextual differences associated with type error corrections. Based on the mined generalized fix templates, TypeFix generates code prompts for pre-trained code models, utilizing the mined templates as domain-specific knowledge. This approach significantly improves the performance of prompt-based program repair by allowing for the adaptive placement of masks tailored to a certain type error, rather than relying on randomly determined locations.

Fourthly, we undertake an empirical analysis of existing approaches to API recommendation. Recognizing that third-party packages in runtime environments primarily furnish implementations for external APIs, we believe that recommending high-quality external APIs can mitigate some runtime environment conflicts. To this end, we evaluate 11 existing API recommendation approaches alongside four widely-used Integrated Development Environments (IDEs), and construct a benchmark, designated APIBench. Utilizing APIBench, we extract actionable insights and identify prevailing challenges of API recommendation. This study

contributes to a deeper understanding and improvement of API recommendation systems, which is crucial for developing reliable Python software.

Lastly, we introduce a novel source-level runtime environment conflict detection approach named PyConf. Previous approaches primarily addressed conflicts among third-party packages within the runtime environment but overlooked potential conflicts between these packages and the software itself. PyConf implements three targeted checks at distinct stages of library interaction: setup, packing, and usage. This approach systematically examines the compatibility of the constructed runtime environments with the software by monitoring the execution of import statements and detecting possible run-time errors. Through the deployment of PyConf on the PyPI platform, we have identified fifteen kinds of configuration issues and discovered that 183,864 library releases have potential configuration problems. This method significantly advances our capability to foresee and mitigate runtime environment conflicts, enhancing the stability and reliability of software deployments.

In summary, this thesis addresses the reliability issues arising from the dynamic type system and the dynamic runtime environment associated with the Python language. Our primary focus is on devising methodologies to prevent, detect, and rectify these reliability issues. We have proposed a comprehensive suite of approaches tailored to enhance the run-time reliability of Python software. Extensive experimental evaluations underscore the effectiveness of the approaches introduced in this thesis, demonstrating significant advancements in managing the inherent challenges posed by the dynamic features of Python.

論文題目: 智能化 Python 軟件運行時可靠性分析

作者　　: 彭昀

學校　　: 香港中文大學

學系　　: 計算機科學與工程學系

修讀學位: 哲學博士

摘要　　:

Python 近來成為一種流行的動態編程語言，自 2019 年以來在 GitHub 的使用量位居第二。Python 開發者從 Python 的動態特性中受益匪淺。其中最重要的動態特性包括動態類型系統和動態運行時環境。動態類型系統省去了代碼中類型聲明的需求，從而極大地簡化了開發者編寫泛型函數的工作。動態運行時環境消除了開發階段將軟件和整個運行時環境打包編譯的需要，使得 Python 擁有豐富的第三方函式庫支持。得益於動態特性帶來的快速原型的優勢，開發者在構建 Python 應用時可以投入更少的努力。

然而，動態特性的代價在於可能帶來更多運行時問題，這威脅到了 Python 軟件的可靠性。由於缺少嚴格的規範，如類型聲明和軟件分發前的編譯，Python 軟件在使用中面臨更多的可靠性問題，例如類型錯誤和運行時環境衝突。在本論文中，我們提出了一種檢測和解決由於 Python 的動態類型系統和動態運行時環境所引發的可靠性問題的方法。

首先，我們提出了一種名為 HiTyper 的混合類型推斷方法，基於靜態類型推斷和神經網絡預測。類型推斷在靜態上為代碼中的變量提供類型，這可以進一步用於通過類型檢查工具檢查潛在的類型衝突來檢測類型錯誤。我們希望結合靜態類型推斷中的類型正確性優勢和神經網絡預測的有效性及高覆蓋率。為實現這一目標，我們記錄每個函數中變量之間的類型依賴關係，並將依賴信息編碼到類型依賴圖（TDGs）中。基於類型依賴圖，我們可以輕鬆地在節點中整合類型推斷規則以進行靜態推斷，並使用類型拒絕規則來檢查神經網絡預測的正確性。HiTyper 會迭代進行靜態推斷和神經網絡預測，直到類型依賴圖完全

被推斷出來，從而獲得代碼中所有變量的類型。

其次，我們提出了一種名為 TypeGen 的生成型類型推斷方法，以進一步提升 HiTyper 的性能。我們認識到 HiTyper 的性能上限依賴於它所使用的深度學習模型的表現，因此我們基於強大的大型語言模型設計了一種更先進的生成方法。TypeGen 通過將靜態分析的類型推斷步驟轉化為基於類型依賴圖的提示，創建思維鏈（COT）提示詞，使語言模型能夠從靜態分析推斷類型的過程中學習。TypeGen 結合思維鏈提示詞、代碼片段和類型提示，從少量人工類型標註中構建示例提示，教導語言模型通過上下文學習生成類似的思維鏈提示詞，生成的思維鏈提示詞中包含了我們所需的最終類型預測。

第三，我們還提出了一種基於領域認識和提示詞的程序修復方法，名為 TypeFix ，用於修復現有的 Python 類型錯誤。基於提示詞的程序修復方法相當有效，但其性能依賴於用於添加掩碼的提示詞模板的質量。為創建有領域認識的提示詞模板，TypeFix 首先通過一種新穎的層次聚類算法挖掘出通用的修復模板。這些識別出的修復模板顯示了現有類型錯誤修復的常見編輯模式和上下文。然後 TypeFix 利用這些通用的修復模板作為領域知識，為代碼預訓練模型生成代碼提示，在其中掩碼的位置是根據每個類型錯誤自適應定位的，從而大大提高了基於提示詞的程序修復方法的性能。

第四，我們對現有的 API 推薦方法進行了實證研究。基於對於運行時環境中大多數第三方包用於為軟件中調用的外部 API 提供實現的洞察，我們認為推薦高質量的外部 API 可以防止運行時環境衝突。我們研究了 11 種現有方法和 4 種廣泛使用的 IDE，並建立了一個名為 APIBench 的基準測試集。基於 APIBench ，我們提煉出了一些可行的研究方向和 API 推薦面臨的挑戰。我們還總結了一些對提高 API 推薦方法性能的啟示和方向。

最後，我們建立了一種名為 PyConf 的源代碼級別運行時環境衝突檢測方法。先前的方法僅關注運行時環境中第三方包之間的衝突，卻忽略了第三方包與軟件之間的衝突。PyConf 採用三種不同的檢查，分別針對第三方庫的設置、打包和使用階段。PyConf 通過追蹤軟件執行中的導入語句來識別潛在的運行

時錯誤，檢查建立的運行時環境與軟件的兼容性。基於 PyConf ，我們識別出 15 種配置問題，並發現 PyPI 有 183,864 個第三方庫版本受到潛在配置問題的影響。

總結來說，本論文針對由於 Python 語言的動態類型系統和動態運行時環境所導致的可靠性問題。我們專注於如何預防、檢測和修復可靠性問題，並提出了一系列方法。廣泛的實驗證明了我們在本論文中提出的方法的有效性。

Acknowledgement

I would like to extend my deepest gratitude to my PhD supervisor, Professor Michael R. Lyu, for his invaluable support and guidance throughout my doctoral studies. His mentorship has been instrumental not only in shaping my research skills but also in broadening my understanding of the world and enhancing my enjoyment of life. I consider myself extraordinarily fortunate to have had Professor Lyu as my supervisor.

Secondly, I am deeply appreciative of the time and effort my thesis committee members, Professors Qiang Xu, Wei Meng, and Lingming Zhang, have devoted to evaluating and improving my thesis. Their insights and suggestions have been greatly beneficial.

Thirdly, I owe a profound debt of gratitude to my family, whose unwavering support has been indispensable throughout my PhD journey. The challenges of the past four years would have been insurmountable without their constant encouragement and support.

Fourthly, I extend my heartfelt thanks to Professor Cuiyun Gao, who provided invaluable guidance at the onset of my PhD studies. From selecting a research topic to writing research papers, Professor Gao' s mentorship taught me crucial strategies for developing novel methodologies and crafting effective research papers. Her guidance laid the foundation for my subsequent work during the PhD program.

I am also fortunate to have worked alongside many exceptional research fellows and senior group members. I am grateful for the collaboration and significant contributions of Yintong Huo, Wenxuan Wang, Shuzheng Gao, Chaozheng Wang, Shuqing Li, Yichen Li, Wenwei Gu, Jen-tse Huang, as well as our esteemed alumni Qirun Zhang and Xiaoxue Ren. Their cooperation on various research projects and their insightful suggestions have greatly enriched the content of this thesis.

I am also pleased to acknowledge my group members: Jinyang Liu, Jianping Zhang, Jiacheng Shen, Shuyao Jiang, Baitong Li, Zhihan Jiang, Renyi Zhong, Jinxi Kuang, Yizhan Huang, and Yuxuan Wan. It has been a pleasure to spend the past four years collaborating and growing together.

Additionally, I am grateful to all those who have directly or indirectly offered helpful suggestions or assistance during my PhD studies. Their support played a crucial role in helping me navigate and overcome the numerous challenges I encountered.

Last, but most importantly, I would like to thank myself. Completing this demanding journey in academia, facing various challenges in both scholarly pursuits and personal life, and achieving notable success in the field of software engineering as a PhD candidate are accomplishments that merit recognition. Thank you to myself for persevering through these four transformative years.

Again, thank you all!

To my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents our research towards run-time reliability engineering for Python software with a focus on the reliability issues brought by the dynamic type system and run-time environment in Python. Given the great popularity and fast development of Python language, the topic of this thesis is an important field of study and practice in software analysis and reliability. We provide a brief overview of research problems under study in Sec. 1.1, and highlight the main contributions of this thesis in Sec. 1.2. The overall structure of this thesis is described in Sec. 1.3.

1.1 Overview

Python, recognized for its flexibility as a dynamic programming language, has seen a significant surge in popularity. According to data from GitHub Octoverse [56], which monitors trends in open-source software, Python ascended to the position of the second most utilized programming language in 2019. Presently, Python exhibits an annual growth rate exceeding 20% in its contributor base, underscoring its expanding adoption and potential within the developer commu-

nity. This trend highlights Python's increasing relevance and utility in modern software development.

Python is renowned for its ease of learning and usability. Its powerful dynamic features facilitate fast prototyping, significantly enhancing software development efficiency. Among these features, the *dynamic type system* and *dynamic run-time environment* are particularly crucial. The dynamic type system allows variable types to be determined at run-time, enabling developers to effortlessly craft generic functions. The dynamic run-time environment obviates the need for compiling Python projects into binary code, so developers only need to provide the core scripts and associated configuration files. Python's package management tools, such as `pip`, will analyze these configurations to build run-time environments that incorporate all necessary dependencies, thus streamlining code execution. Leveraging the dynamic run-time environments, developers can seamlessly integrate external APIs into their projects by specifying dependencies in the configuration files. This flexibility has spurred the proliferation of diverse third-party packages, minimizing the effort to replicate functionalities. Presently, PyPI [41], the largest third-party package management platform for Python, hosts over 530,000 packages with more than 5,640,000 releases, significantly bolstering Python's widespread adoption.

Run-time Reliability Issues. The flexibility afforded by Python's dynamic type system and dynamic run-time environment does not come without costs. These features, while facilitating rapid development, introduce more reliability issues compared to static programming languages like Java and C++. Since Python software is not compiled before distribution, it bypasses the rigorous checks and optimizations typically performed during the compilation process of static languages. As a result, when software is deployed, it may contain latent issues that could disrupt normal operation. Key among these are *type errors*, which

arise from Python's dynamic type system, and *run-time environment dependency conflicts*, where incorrect run-time environments can interfere with the software. These issues contribute to potential instability and complicate the maintenance of software reliability.

Type errors manifest when the types of variables involved in an operation are incompatible. For instance, attempting to access a `key` in a variable of type `List` results in a type error, as `List` lacks the `key` attribute inherent to `Dict` types. In Python, where developers are not obligated to specify variable types explicitly, the likelihood of overlooking potential type conflicts increases, leading to type errors during code execution. A recent study [143] indicates that over 30% of discussions on Python in forums such as Stack Overflow and GitHub pertain to type errors. Further analysis reveals that approximately half of these errors require more than one week to resolve. This extended troubleshooting duration imposes significant burdens on developers, underscoring the challenges inherent in maintaining Python software effectively.

Run-time environment dependency conflicts arise when the version of a dependency required by software does not match the version present in the run-time environment. For example, if a software's API demands a version of `torch` greater than 2.2.0 (`torch>2.2.0`), but the installed version is `torch==2.0.0`, a conflict occurs. This mismatch can lead to errors when the API is invoked, which is also a common issue highlighted in developer discussions. The run-time environment, encompassing the operating system, system libraries, the interpreter, and third-party packages, is integral to the smooth execution of software. Since Python's package management platforms do not verify the correctness of configuration files provided by developers, and given that dependent third-party packages may evolve independently of the software, conflicts frequently emerge. Users who set up their run-time environments based on the original configuration files often

encounter these conflicts, which can significantly disrupt software functionality.

Challenges. Significant research has been dedicated to resolving the reliability issues associated with Python's dynamic type system. Two primary approaches have emerged to mitigate type errors.

The first approach involves introducing static typing into Python code through type inference techniques, allowing traditional type-checking tools to identify potential type errors. Initially, this approach utilized rule-based type inference methods similar to those employed in compilers for static languages. However, these methods often struggle with Python's lack of explicit type declarations, particularly with function arguments and other variables whose types cannot be readily inferred. As a result, coverage was typically limited. More recent efforts have shifted towards data-driven techniques, employing deep learning models to treat code as text and predict variable types. Although these models can offer type predictions for each variable, they cannot guarantee the accuracy of these predictions due to their probabilistic nature.

The second approach does not prevent new type errors but focuses on fixing existing ones using program repair techniques. These methods utilize manually defined rules and templates to repair specific type errors. While effective for certain scenarios, the need to manually craft these rules and templates makes this approach labor-intensive and less adaptable to type errors not covered by the existing rules.

To tackle the issue of run-time environment conflicts in Python, researchers have also developed two main approaches.

The first approach involves scrutinizing the configuration files provided by developers. This method effectively identifies potential conflicts between dependencies specified in the configuration files. However, simply checking versions does not fully ensure compatibility between the run-time environment and the

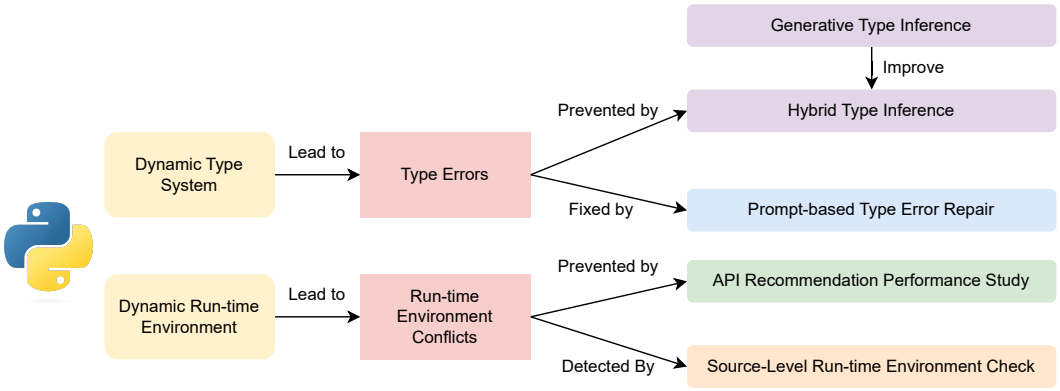


Figure 1.1: The overview of research in the thesis.

Python software. This is because conflicts can also arise from discrepancies between the external APIs used in the software and the versions of dependencies installed in the run-time environment. For instance, conflicts can occur if the software uses an API that has been deprecated, but the version-level checks in configuration files cannot catch this issue.

The second approach focuses on API recommendation. This method assumes that most dependencies in the run-time environments are intended to support external API implementations in the software. By ensuring correct API usage, this approach can significantly reduce run-time environment conflicts. For example, avoiding the use of deprecated APIs would prevent the aforementioned conflict. API recommendation tools are designed to help developers select appropriate APIs, thus saving time and reducing errors. However, the effectiveness of these tools is not well-studied, indicating that further exploration and validation are needed in this area.

Existing approaches offer valuable strategies for mitigating type errors and run-time environment conflicts, yet each has its limitations and requires further refinement to mitigate the reliability issues.

Our Research. In this thesis, we study the run-time reliability issues caused

by Python's dynamic type system and dynamic run-time environment. Figure 1.1 provides an overview of the research scope. Our research seeks to significantly improve the reliability of Python software by addressing two of the most critical challenges in dynamic programming environments: type errors and run-time environment conflicts.

For type errors, we introduce a hybrid type inference method that combines the correctness of static type inference with the effectiveness of data-driven techniques. This approach enhances the accuracy of type inference by integrating the methodological soundness of static analysis with the effectiveness afforded by data-driven models. Additionally, we innovate a generative type inference technique to further improve the performance of data-driven approaches. To fix existing type errors, we propose a novel domain-aware prompt-based repair technique utilizing the capabilities of advanced large language models.

For run-time environment conflicts, we begin with a thorough empirical analysis of existing API recommendation approaches, evaluating their effectiveness and summarizing key insights for developing approaches that can accurately recommend high-quality APIs compatible with the required run-time environments. Following this, we introduce a fine-grained, source-level detection method for identifying conflicts between Python software and its run-time environment. This approach aims to pinpoint and resolve discrepancies that can lead to run-time failures.

1.2 Thesis Contributions

We summarize the contributions of this thesis as follows.

1. **Hybrid Type Inference Based on Static Inference and Deep Learning.**

Type inference for dynamic programming languages such as Python is an important yet challenging task. Static type inference techniques can precisely infer variables with enough static constraints but are unable to handle variables with dynamic features. Deep learning (DL) based approaches are feature-agnostic, but they cannot guarantee the correctness of the predicted types. Their performance significantly depends on the quality of the training data (i.e., DL models perform poorly on some common types that rarely appear in the training dataset). It is interesting to note that the static and DL-based approaches offer complementary benefits. Unfortunately, to the best of our knowledge, precise type inference based on both static inference and neural predictions has not been exploited and remains an open challenge. In particular, it is hard to integrate DL models into the framework of rule-based static approaches. We fill the gap and propose a hybrid type inference approach named HiTyper based on both static inference and deep learning. Specifically, our key insight is to record type dependencies among variables in each function and encode the dependency information in type dependency graphs (TDGs). Based on TDGs, we can easily integrate type inference rules in the nodes to conduct static inference and type rejection rules to inspect the correctness of neural predictions. HiTyper iteratively conducts static inference and DL-based prediction until the TDG is fully inferred. Experiments on two benchmark datasets show that HiTyper outperforms state-of-the-art DL models by exactly matching 10% more human annotations. HiTyper also achieves an increase of more than 30% on inferring rare types. Considering only the static part of HiTyper, it infers $2\times \sim 3\times$ more types than existing static type inference tools. Moreover, HiTyper successfully corrected seven wrong human annotations in six GitHub projects, and two of them have already been approved by the

repository owners.

2. **Generative Type Inference.**

   The previous hybrid type inference we proposed takes advantage of static type inference and deep learning models. However, we find that the performance upper bound of hybrid type inference relies on the performance of the deep learning models we choose. How to improve the performance of deep learning models remains a great challenge. Current type inference approaches based on deep learning models are supervised type inference approaches, which require large, high-quality annotated datasets to train the models and are limited to pre-defined types. Based on powerful pre-trained deep learning models, the cloze-style approaches reformulate the type inference problem into a fill-in-the-blank problem by leveraging the general knowledge in powerful pre-trained code models. However, their performance is limited since they ignore the domain knowledge from static typing rules which actually reflect the inference logic. Furthermore, their predictions are not interpretable, hindering developers' understanding and verification of the results. To address these challenges, we introduce TypeGen, a few-shot generative type inference approach that incorporates static domain knowledge from static analysis. TypeGen creates chain-of-thought (COT) prompts by translating the type inference steps of static analysis into prompts based on the type dependency graphs (TDGs), enabling language models to learn from how static analysis infers types. By combining COT prompts with code slices and type hints, TypeGen constructs example prompts from human annotations. TypeGen only requires very few annotated examples to teach language models to generate similar COT prompts via in-context learning. Moreover, TypeGen enhances the interpretability of results through the use of the *input-explanation-output*

strategy, which generates both explanations and type predictions in COT prompts. Experiments show that TypeGen outperforms the best baseline Type4Py by 10.0% for argument type prediction and 22.5% in return value type prediction in terms of top-1 Exact Match by using only five examples. Furthermore, TypeGen achieves substantial improvements of 27% to 84% compared to the zero-shot performance of large language models with parameter sizes ranging from 1.3B to 175B in terms of top-1 Exact Match.

3. **Domain-aware Prompt-based Type Error Repair.**

   There exist rule-based approaches for automatically repairing Python type errors. The approaches can generate accurate patches for the type errors covered by manually defined templates, but they require domain experts to design patch synthesis rules and suffer from low template coverage of real-world type errors. Learning-based approaches alleviate the manual efforts in designing patch synthesis rules and have become prevalent due to the recent advances in deep learning. Among the learning-based approaches, the prompt-based approach which leverages the knowledge base of code pre-trained models via pre-defined prompts, obtains state-of-the-art performance in general program repair tasks. However, such prompts are manually defined and do not involve any specific clues for repairing Python type errors, resulting in limited effectiveness. How to automatically improve prompts with the domain knowledge for type error repair is challenging yet under-explored. We present TypeFix, a novel prompt-based approach with fix templates incorporated for repairing Python type errors. TypeFix first mines generalized fix templates via a novel hierarchical clustering algorithm. The identified fix templates indicate the common edit patterns and contexts of existing type error fixes. TypeFix then generates

code prompts for code pre-trained models by employing the generalized fix templates as domain knowledge, in which the masks are adaptively located for each type error instead of being pre-determined. Experiments on two benchmarks, including BugsInPy and TypeBugs, show that TypeFix successfully repairs 26 and 55 type errors, outperforming the best baseline approach by 9 and 14, respectively. Besides, the proposed fix template mining approach can cover 75% of developers' patches in both benchmarks, increasing the best rule-based approach PyTER by more than 30%.

4. **Empirical Analysis of API Recommendation.**

   Application Programming Interfaces (APIs), which encapsulate the implementation of specific functions as interfaces, greatly improve the efficiency of modern software development. As the number of APIs grows fast nowadays, developers can hardly be familiar with all the APIs and usually need to search for appropriate APIs for usage. So lots of efforts have been devoted to improving the API recommendation task. However, it has been increasingly difficult to gauge the performance of new models due to the lack of a uniform definition of the task and a standardized benchmark. For example, some studies regard the task as a code completion problem, while others recommend relative APIs given natural language queries. To reduce the challenges and better facilitate the goal of recommending high-quality APIs, we revisit the API recommendation task and aim at benchmarking the approaches. Specifically, we group the approaches into two categories according to the task definition, i.e., query-based API recommendation and code-based API recommendation. We study 11 recently proposed approaches along with 4 widely-used IDEs. One benchmark named APIBench is then built for the two respective categories of approaches. Based on APIBench, we distill some actionable insights and challenges for API recommendation. We also

achieve some implications and directions for improving the performance of recommending APIs, including appropriate query reformulation, data source selection, low resource setting, user-defined APIs, and query-based API recommendation with usage patterns.

5. **Source-Level Run-time Environment Conflict Detection.**

   Python's popularity is largely owing to the extensive support from diverse third-party libraries within the PyPI ecosystem. Nevertheless, the utilization of third-party libraries can potentially lead to conflicts in dependencies, prompting researchers to develop dependency conflict detectors. Moreover, endeavors have been made to automatically infer dependencies. These approaches focus on version-level checks and inference, based on the assumption that configurations of libraries in the PyPI ecosystem are correct. However, our study reveals that this assumption is not universally valid, and relying solely on version-level checks proves inadequate in ensuring compatible run-time environments. We conduct an empirical study to comprehensively study the configuration issues in the PyPI ecosystem. Specifically, we propose PyConf, a source-level detector, for detecting potential configuration issues. PyConf employs three distinct checks, targeting the setup, packing, and usage stages of libraries, respectively. To evaluate the effectiveness of the current automatic dependency inference approaches, we build a benchmark called VLibs, comprising library releases that pass all three checks of PyConf. We identified 15 kinds of configuration issues and found that 183,864 library releases suffer from potential configuration issues. Remarkably, 68% of these issues can only be detected via the source-level check. Our experiment results show that the most advanced automatic dependency inference approach, PyEGo, can successfully infer dependencies for only 65% of library releases. The primary failures

stem from dependency conflicts and the absence of required libraries in the generated configurations. Based on the empirical results, we derive six findings and draw two implications for open-source developers and future research in automatic dependency inference.

1.3 Thesis Organization

The rest of this thesis is organized as follows.

- **Chapter 2**

  In this chapter, we conduct a background review of the dynamic type system and the dynamic run-time environment of Python. In particular, we first introduce the Python type system and some examples of type errors in Sec. 2.1. We then provide an overview of the run-time environment and some examples of run-time environment conflicts in Sec. 2.2. In Sec. 2.3, we review previous work on type inference, program repair, API recommendation, run-time environment conflict detection, and run-time environment dependency inference.

- **Chapter 3**

  In this chapter, we present HiTyper, the first hybrid type inference approach based on static type inference and deep learning models. In Sec. 3.1, we briefly introduce the challenges of previous type inference approaches and the insight behind our hybrid type inference approach HiTyper. We then give a detailed motivating example to show the effectiveness and advantages of HiTyper in Sec. 3.2, compared with previous rule-based and supervised type inference approaches. We present the technical details of HiTyper in Sec. 3.3 and further evaluate HiTyper with selected previ-

ous approaches as baselines in Sec. 3.4. Finally, we list the limitations of HiTyper in Sec. 3.5 and summarize the chapter in Sec. 3.6.

- **Chapter 4**

  In this chapter, we present TypeGen, the first few-shot generative type inference approach. In Sec. 4.1, we briefly introduce the challenges of previous rule-based, supervised, and cloze-style type inference approaches. We then provide the technical details of TypeGen in Sec. 4.2. In Sec. 4.3, we describe the datasets, baselines, metrics used in the evaluation, and the implementation details of TypeGen. We evaluate the performance of TypeGen compared with previous rule-based, supervised, and cloze-style approaches in Sec. 4.4. In Sec. 4.5, we provide an example to demonstrate the interpretability of TypeGen and list the potential limitations of TypeGen. Sec. 4.6 acts as a summary of the entire chapter.

- **Chapter 5**

  In this chapter, we present TypeFix, the first domain-aware prompt-based approach for repairing Python type errors. In Sec. 5.1, we emphasize the importance of handling type errors and introduce the challenges of existing program repair approaches on repairing type errors. In Sec. 5.2, we provide a detailed motivating example to show how domain-aware prompts can help current prompt-based program repair methods repair type errors. Sec. 5.3 presents the technical details of TypeFix. In Sec. 5.4, we describe the datasets, baselines, metrics used in the evaluation, and the implementation details of TypeFix. We evaluate the effectiveness of TypeFix, compared with current rule-based and prompt-based approaches in Sec. 5.5. Finally, We summarize this chapter in Sec. 5.6.

- **Chapter 6**

In this chapter, we conduct an empirical analysis on existing API recommendation approaches and build a benchmark named APIBench. In Sec. 6.1, we introduce the motivation of our empirical analysis and classify current API recommendation approaches into two categories: query-based approaches and code-based approaches. We then provide a background review for query-based and code-based API recommendation approaches in Sec. 6.2. In Sec. 6.3, we describe how we build the benchmark APIBench. Sec. 6.4 and Sec. 6.5 present the empirical results for query-based API recommendation approaches and code-based API recommendation approaches, respectively. In Sec. 6.6, we conclude some implications to further research on how to recommend high-quality APIs. We list the potential threats to the validity of our empirical analysis in Sec. 6.7. Sec. 6.8 summarizes the entire chapter.

- **Chapter 7**

  In this chapter, we present PyConf, the first source-level Python run-time environment conflict detection approach. In Sec. 7.1, we introduce the challenges of current version-level run-time environment conflict detection approaches and provide a motivating example. In Sec. 7.2, we present the technical details of PyConf and how we perform an empirical analysis on popular packages in PyPI based on PyConf. We describe the benchmarks, baselines, metrics, and environments used in the evaluation in Sec. 7.3. In Sec. 7.4, we analyze the dependency conflicts detected by PyConf and summarize the potential reasons. Based on the detected conflicts, we provide several suggestions for developers to avoid run-time environment conflicts in Sec. 7.5. Sec. 7.6 summarizes the entire chapter.

- **Chapter 8**

In this chapter, we summarize the contents of this thesis in Sec. 8.1 and propose some future directions for research on the reliability issues brought by the dynamic type system and the dynamic run-time environment of Python in Sec. 8.2.

Chapter 2

Background

This chapter introduces the dynamic type system and the dynamic run-time environment of Python. For each, we first present the basic definition and then give some examples of type errors, which are brought by the dynamic type system, and run-time environment conflicts, which are derived from the dynamic run-time environments. Furthermore, we provide a literature review of related research efforts in mitigating the reliability issues brought by them, including type inference, automatic program repair, API recommendation, run-time environment conflict detection, and run-time environment dependency inference.

2.1 Dynamic Type System

The dynamic type system is a typical feature that distinguishes Python from static languages such as C++ and Java. In this section, we provide the definition of the Python type system and present some examples of type errors that occurred under the dynamic type system of Python.

$$\theta \in \textit{Type}\ (\Theta) ::= \gamma \mid \alpha[\theta, ..., \theta] \mid u \mid \textbf{None} \mid \textbf{type}$$

$$\gamma \in \textit{Elementary Type}\ (\Gamma) ::= \textbf{int} \mid \textbf{float} \mid \textbf{str} \mid \textbf{bool} \mid \textbf{bytes}$$

$$\alpha \in \textit{Generic Type}\ (A) ::= \textbf{List} \mid \textbf{Tuple} \mid \textbf{Dict} \mid \textbf{Set} \mid \textbf{Callable} \mid \textbf{Generator} \mid \textbf{Union}$$

$$b \in \textit{Builtin Type}\ (B) ::= \gamma \mid \alpha[\theta]$$

$$u \in \textit{User Defined Type}\ (U) ::= \textit{all classes and named tuples in code}$$

$$o \in \textit{Overloading User Defined Type}\ (O) ::= \textit{all classes with operator overloading in code}$$

Figure 2.1: Types in Python.

2.1.1 Python Type System

Fig. 2.1 presents the classification of different types as defined in the official Python documentation [37] and its associated type checker, `mypy` [36]. Notably, we have excluded the `object` and `Any` types due to their non-conformance with strict static typing principles. Types can generally be divided into two categories: built-in types, which are predefined within the Python language specification, and user-defined types, which are crafted by developers. User-defined types allow developers to specify the operations or methods that the types support. This capability extends to overriding certain built-in operations. For instance, a developer might implement an `__add__()` method within a class, enabling direct addition of instances derived from this class via the built-in `+` operator. This is known as *operator overloading*. We introduce a subcategory specifically for user-defined types that demonstrate operator overloading since they have different

typing rules.

The type categories shown in Fig. 2.1 are widely used in most static type inference techniques [133, 160, 166]. Differently, deep learning-based studies [7, 129] generally categorize the types into *common types* and *rare types* based on a pre-defined threshold of occurrence frequencies (e.g., 100 in [7]). For a fair comparison, we also follow this definition for evaluation.

We show the expressions for typing in Python in Fig. 2.2. Expressions in Fig. 2.2 generate new types based on existing types in operands. For example, the boolean expression "v_1 *And* v_2" takes the types of variables v_1 and v_2 as inputs and outputs a result with type `bool`. Note that the statements in the Python language specification also have types, but we do not consider them in this thesis because they hardly invoke type errors and are used in certain static analyses.

With the definition of types and the definitions of expressions that generate new types, we can add typing rules for specific expressions to guide type inference, which aims to give a type for every variable and expression in the code. However, the dynamic type system of Python does not require the type declaration when defining a new variable. This means that we cannot statically get the types of variables in code and can only do this at run-time when all variables and expressions have specific values. Therefore, the built-in type inference in Python interpreter only performs at run-time and invokes type errors when it identifies type conflicts.

2.1.2 Python Type Errors

Python type errors arise when the Python interpreter fails to determine a compatible type assignment for the variables and expressions within a code segment. Upon encountering such an error, the Python interpreter will raise a `TypeError` and halt the execution of the code. Consequently, type errors are

$$
\begin{aligned}
e \in \mathit{Expr} ::= {} & v \mid c \mid e\ \textbf{blop}\ e \mid e\ \textbf{numop}\ e \mid \\
& e\ \textbf{cmpop}\ e \mid e\ \textbf{bitop}\ e \mid \\
& (e, ..., e) \mid [e, ..., e] \mid \\
& \{e : e, ..., e : e\} \mid \{e, ..., e\} \mid \\
& [e\ \textbf{for}\ e\ \textbf{in}\ e] \mid \{e\ \textbf{for}\ e\ \textbf{in}\ e\} \mid \\
& \{e : e\ \textbf{for}\ e, e\ \textbf{in}\ e\} \mid (e\ \textbf{for}\ e\ \textbf{in}\ e) \mid \\
& e(e, ..., e) \mid e[e : e : e] \mid e.v \\
v \in \mathit{Variables} ::= {} & \textit{all identifiers in code} \\
c \in \mathit{Constants} ::= {} & \textit{all literals in code} \\
\mathit{blop} \in \mathit{Boolean\ Operations} ::= {} & \mathit{And} \mid \mathit{Or} \mid \mathit{Not} \\
\mathit{numop} \in \mathit{Numeric\ Operations} ::= {} & \mathit{Add} \mid \mathit{Sub} \mid \mathit{Mult} \mid \mathit{Div} \mid \mathit{Mod} \mid \\
& \mathit{UAdd} \mid \mathit{USub} \\
\mathit{bitop} \in \mathit{Bitwise\ Operations} ::= {} & \mathit{LShift} \mid \mathit{RShift} \mid \mathit{BitOr} \mid \mathit{BitAnd} \mid \\
& \mathit{BitXor} \mid \mathit{FloorDiv} \mid \mathit{Invert} \\
\mathit{cmpop} \in \mathit{Compare\ Operations} ::= {} & \mathit{Eq} \mid \mathit{NotEq} \mid \mathit{Lt} \mid \mathit{LtE} \mid \mathit{Gt} \mid \mathit{GtE} \mid \\
& \mathit{Is} \mid \mathit{IsNot} \mid \mathit{In} \mid \mathit{NotIn}
\end{aligned}
$$

Figure 2.2: The syntax of expressions for typing in Python

critical, as they pose a significant threat to the reliability of Python software, potentially leading to the abrupt termination of applications.

We give some examples of type errors in Listing. 2.1.

```
#Example 1
int_value = 100
str_value = "10"
```

```
result = int_value / str_value
#Example 2
def divide(input_list):
    result = input_list[0]
    for item in input_list[1:]:
        result = result / item
    return result
divide(100)
divide([100, "10"])
```

Listing 2.1: Examples of type errors in Python.

In the first example, the variable *int_value* is assigned the type `int`, while *str_value* is assigned the type `str`. The variable *result* attempts to store the outcome of a division operation between *int_value* and *str_value*. However, such an operation is undefined between an `int` type and a `str` type, leading the Python interpreter to be unable to determine the expression's type and consequently, it throws a type error. This error is straightforward and can be statically detected by type checkers such as MyPy [133].

The second example illustrates a more complex scenario involving type errors. In the function from this example, division is performed on elements within an input list. Due to Python's dynamic typing, where developers are not required to specify type declarations, the type of the function argument *input_list* remains unknown until the function is executed. This ambiguity makes the function particularly susceptible to type errors.

We demonstrate this with two function calls that result in type errors. In the first call, the function receives an integer value 100 as its parameter. Since integers are not iterable, the Python interpreter raises a type error. In the second call, the function is passed a list containing both an integer and a string. Here, the interpreter throws a type error because the division between an `int` type and

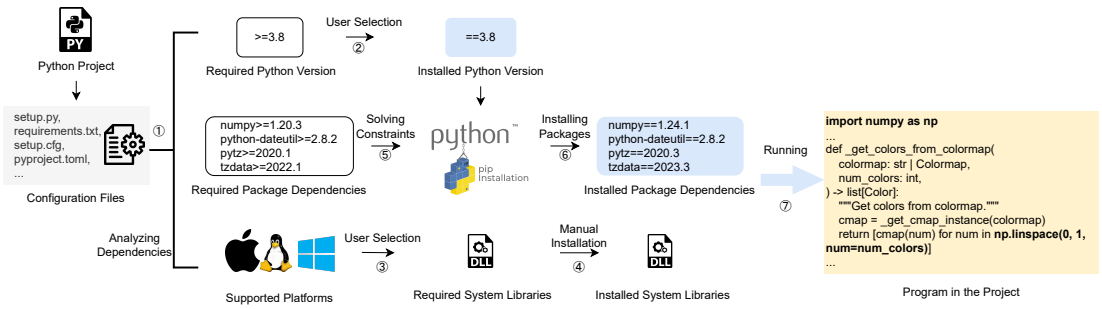


Figure 2.3: The typical process of run-time environment installation for Python projects.

a `str` type is undefined, as illustrated in line 9 of the code.

Writing complex functions without explicit type declarations poses significant challenges for developers, who must anticipate and manage potential type inconsistencies. Although static type checking tools like MyPy can aid in detecting some static type errors, their effectiveness is limited without explicit type declarations, restricting their ability to preemptively identify and resolve type errors in the code.

2.2 Dynamic Run-time Environment

2.2.1 Python Run-time Environment

As an interpreted programming language, Python offers the advantage of not requiring compilation prior to execution. This attribute facilitates fast prototyping and enables Python programs to be executed on different platforms. However, benefiting from rich support from external libraries, nearly all Python projects depend on multiple third-party or system libraries to avoid redundant implementations of common functionalities. Therefore, it becomes imperative to establish the appropriate run-time environment, comprising all necessary libraries, before

running a Python project effectively.

We illustrate the process of installing the run-time environment for a Python project based on its source code in Fig.2.3. Typically, developers document all project dependencies in configuration files. As the Python community evolves, various configuration formats like `requirements.txt` and `setup.py` have emerged. Additionally, there are diverse developer tools, such as `setuptools`, available to analyze these configuration files (① in Fig.2.3). The configuration files contain three types of dependencies: 1) the required Python version, 2) the necessary third-party libraries, and 3) the required platform and corresponding system libraries.

In most cases, users must first select the appropriate Python version and platform (② and ③ in Fig.2.3) before proceeding with the installation of other dependencies. If the Python project relies on some system libraries of the selected platform, users may also need to install them manually (④ in Fig. 2.3). Since Python third-party libraries are hosted on the PyPI platform [41], Python Software Foundation also provides a dedicated tool named `pip` [39] to facilitate automated installation. `Pip` first resolves the constraints of third-party libraries provided in the configuration file (⑤ in Fig. 2.3), and then selects the latest valid version for each library (⑥ in Fig. 2.3). By ensuring the presence of the appropriate Python version, third-party libraries, and system libraries, users can successfully execute certain Python projects (⑦ in Fig. 2.3).

Through the above process of building run-time environments, we can see that most efforts are paid on the user side and developers only provide configuration files to guide the construction of run-time environments. As the relied third-party packages continue to evolve, different users may build different run-time environments even if they are given the same configuration files. This is quite different from what static programming languages such as C++ and Java

do. For C++ and Java, developers should prepare the implementations of all relied third-party packages and compile them with the core software into a single binary. Users only need to download the compiled binary and run it in the declared operating system by developers.

2.2.2 Python Run-time Environment Conflicts

The dynamic run-time environment of Python offers substantial convenience for developers in building and distributing software. This flexibility simplifies the deployment process and enhances accessibility across different platforms. However, this feature also introduces potential risks, notably run-time environment conflicts. Run-time environment conflicts exist when users try to build the run-time environments and when users run the software in the built run-time environments.

When configuring run-time environments using configuration files, developers may encounter two primary types of conflicts:

- **Incorrect Configurations.** This issue arises when the configuration files distributed by developers contain errors. A common example is specifying a version of a third-party package that does not exist on the package management platform. Such discrepancies can prevent the successful installation of the required packages, leading to run-time errors.

- **Installation Conflicts.** These occur when the configuration files specify two or more versions of third-party packages that are incompatible and cannot be co-installed. For instance, if a package A requires a dependency $C > 1.0$ and another package B requires $C < 1.0$, there is a conflict between A and B over the acceptable version of C, making it impossible to satisfy both dependencies simultaneously. These are referred to as *version-level*

conflicts.

While significant research efforts focus on detecting and resolving version-level conflicts, the conflicts between software and run-time environments remain less explored. These conflicts arise when the implementations of APIs utilized in the software are absent in the installed run-time environments, leading to execution halts. Python specifically designates a series of errors to signal such conflicts: ImportError, ModuleNotFoundError, and AttributeError. A prevalent form of these conflicts involves the use of deprecated APIs. When software relies on older versions of APIs that have since been deprecated and removed, issues arise, particularly because package management tools like *pip* typically install the most recent versions of packages. If these newer versions no longer support the deprecated APIs, the necessary implementations may not be present in the run-time environment, leading to what we refer to as source-level conflicts.

Detecting source-level conflicts necessitates a thorough analysis of both the software and the run-time environment. This task can be challenging for human developers, given the intricate details involved in assessing API compatibility and version history across different environment setups. Tools and automated processes that can scan and analyze these discrepancies are essential for mitigating the risk of such conflicts, ensuring software reliability and consistency in various deployment contexts.

2.3 Related Work

2.3.1 Type Inference

We classify existing type inference approaches into three categories: rule-based, supervised and cloze-style approaches, and present an overview of three kinds of type inference approaches in Fig. 2.4.

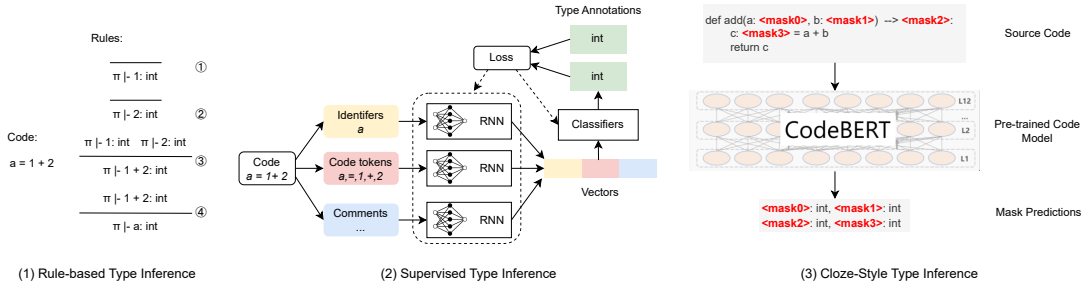


Figure 2.4: Three kinds of type inference approaches.

Rule-based Type Inference. Rule-based approaches for type inference rely on predefined rules to determine the types of variables. Fig. 2.4(1) shows an example where four rules are associated with the type inference of variable *a*. Each rule has premises (above the line) and conclusions (below the line). A rule can be triggered only if all premises are known, and then the result type is given based on the conclusion.

To address the need for static type hints in dynamically typed programming languages, various approaches have been proposed for type inference and checking, such as Pyright [161] and Pylance from Microsoft, Pyre from Meta [160], Pytype from Google [166], and Python's official type checker mypy [133]. In addition to industry tools, some academic approaches have been proposed for type inference in different programming languages, such as Python and JavaScript [10, 20, 29, 46, 80, 151]. While these approaches are quite accurate, they are limited by the low coverage problem caused by dynamic features and external calls [152].

Supervised Type Inference. Supervised type inference approaches utilizing deep learning models have made significant progress in predicting types for dynamic languages. Fig. 2.4(2) illustrates the typical process of these approaches: features are extracted from code and encoded into vectors using deep learning models such as recurrent neural networks (RNNs) [188]. A classifier is then used to classify the vectors into pre-defined types. The loss is calculated

based on the type prediction of the classifier and the human type annotation, and the parameters of the deep learning models and classifier are updated via back-propagation.

Allamanis *et al.* [7] adopt an open vocabulary model which encodes code as graphs to predict types. Pradel *et al.* [157] uses multiple RNN models to encode features such as identifiers and code tokens. Mir *et al.* [129] improve the top-1 accuracy via a deep similarity clustering algorithm. Wei *et al.* [219] propose to use graph neural networks to predict types. Jesse *et al.* [81] propose TypeBERT by reformulating type prediction as a NER problem. Peng *et al.* [153] propose HiTyper, which uses deep learning models to recommend types for static inference. While these approaches achieve satisfying performance, they require high-quality datasets for training, which can be difficult to obtain in the wild. Furthermore, supervised type inference approaches only provide predictions without any explanation about how they infer the types, making it challenging for developers to understand and verify the results.

Cloze-Style Type Inference. To enhance the reliability of Python software and prevent potential type errors, the Python Software Foundation has introduced a series of Python Enhancement Proposals (PEPs)[99, 100, 204, 250] that enable developers to add static type annotations to their code. As these annotations become part of the code, they can be leveraged by pre-trained code models that are trained on a vast amount of open-source Python programs. Cloze-style type inference approaches, as illustrated in Fig.2.4(3), add masks on the locations of type annotations in the code and invoke pre-trained code models to fill in the masks with predicted types.

All pre-trained code models with Masked Language Modeling (MLM) training objectives such as CodeBERT [31], GraphCodeBERT [62] and CodeT5 [217] can be naturally used to predict type annotations. UniXcoder [61] is a unified

cross-modal pre-trained model to support both code-related understanding and generation tasks. InCoder [44] is a large code generative model that can refill arbitrary regions of code. More recently, Llama3 [119] and GPT4 [147] even provide more powerful support for all code-related tasks. Leveraging pre-trained code models, cloze-style type inference approaches can be readily implemented. However, they still exhibit limited performance as they solely rely on the general knowledge of pre-trained code models. These approaches can hardly handle complicated types without domain knowledge from static typing rules, and their predictions lack interpretability without explanations.

2.3.2 Automatic Program Repair

As an important method to improve the reliability of software, automatic program repair (APR) has drawn a lot of attention [52, 131] in recent years. Currently, most APR approaches can be classified into rule-based approaches and learning-based approaches.

Rule-based Approaches. Rule-based APR approaches leverage pre-defined templates and rules to generate patches for bugs via static and dynamic analysis. There are a series of rule-based APR approaches designed for Java programs [51, 55, 85, 118, 138, 223, 233, 236], and for the memory bugs of C programs [48, 70, 74, 98, 238]. For Python programs, PyTER [143] utilizes nine pre-defined templates with type-aware fault localization to repair type errors.

Learning-based Approaches. Learning-based APR approaches become quite popular and demonstrate their superior performance recently. Motivated by the study [210, 211, 239] of neural machine translation (NMT) [197], there are many research efforts being devoted to NMT-based APR approaches. SequenceR [21] presents a sequence-to-sequence LSTM model for program repair. DLFix [104] leverages tree-based RNN to transform code inputs and generate

patches. CoCoNuT [112] separates the context and buggy line in NMT-based APR. CURE [86] is the first approach that integrates pre-trained models in NMT-based APR, followed by work [27, 115]. Recoder [249] generates code edits instead of modified code. RewardRepair [240] uses execution-based backpropagation to improve the compilation rate of patches generated by NMT-based APR approaches. In the era of large language models, AlphaRepair [228] is the first prompt-based APR approach that transforms the APR problem into a fill-in-the-blank problem, and generate patches by filling the masks with tokens predicted by LLMs. Prenner *et al.* [159] propose to use LLMs to directly generate the fixed function based on the identified buggy function. ChatRepair [230] proposes to fix bugs via multiple iterations of chatting with LLMs. Xia *et al.* [227] conducts a comprehensive study on automated program repair methods based on LLMs.

2.3.3 API Recommendation

Query-based API Recommendation. Multiple existing works [18, 60, 76, 106, 109, 117, 132, 167, 172, 196, 199, 232, 234, 242, 244, 246, 248, 248] explore the possibility to provide developers with concrete API recommendation, using the natural language queries as input. Most of these works utilize open-source code bases, and some also use the knowledge in crowd-sourcing forums and wiki websites for augmentation.

Portfolio [117], proposed by McMillan *et al.* , recommends relevant APIs by utilizing several NLP techniques and indexing approaches with spreading activation network (SAN) algorithms as well as PageRank [13]. Zhang *et al.* supplement the call graph with control flow analysis and design Flow-Augmented Call Graph (FACG) to utilize it for API recommendation [244]. Chan *et al.* model API invocations as API graphs and design subgraph search algorithm to recommend APIs [18]. Rahman *et al.* collect the crowdsourced knowledge on Stack Over-

flow to extract keyword-API correlations and find several relevant API classes based on them [172]. Huang *et al.* propose BIKER [76] to bridge the lexical gap and knowledge gap that previous approaches faced during API recommendation. BIKER obtains API candidates from Stack Overflow, and uses the similarity between queries and documentations as well as Stack Overflow posts to recommend API methods. Liu *et al.* propose KG-APISumm [109], which is the first knowledge graph designed for API recommendation. KG-APISumm sorts the APIs through similarity calculation between queries and relevant parts of the constructed knowledge graph to recommend API classes.

Other than coding problems encountered by developers, there is another source of natural language functionality descriptions for APIs: feature requests from product managers or users. Thung *et al.* propose a method to recommend APIs based on the feature requests by learning from other modifications of the projects [199]. Xu *et al.* propose MULAPI [234], which takes feature locations, project repositories and API libraries into consideration when recommending APIs.

DeepAPI [60], proposed by Gu *et al.* , is the first approach that combines deep learning with API recommendation. It reformulates the API recommendation task as a query-API translation problem and uses an RNN Encoder-Decoder model to recommend API sequences. Xiong *et al.* propose to use representation learning to recommend web-based smart service [232]. Ling *et al.* propose GeAPI [106] based on graph embedding to provide more semantic information about and between APIs. GeAPI utilizes projects' source code to automatically construct API graphs and leverages graph embedding techniques for API representation. Given a query, it searches relevant subgraphs on the original graph and recommends them to developers. Zhou *et al.* propose BRAID [248] and utilize approaches such as active learning as well as learning-to-rank based on the

feedback of users to further improve the performance.

Code-based API Recommendation. Zhong *et al.* propose MAPO [247] to mine API usage patterns and then recommend the relevant usage patterns to developers. Schäfer *et al.* propose Pythia [187] to utilize static pointer analysis and usage-based property inference to recommend APIs for JavaScript. Wang *et al.* propose UP-Miner [209] and use source code to extract succinct usage patterns to recommend APIs. Nguyen *et al.* propose APIREC [135], which uses fine-grained code changes and the corresponding changing contexts to recommend APIs. D' Souza *et al.* propose PyReco [28]. It first extracts API usages from open-source projects and uses such information to rank the API recommendation results by utilizing nearest neighbor classifier techniques. Fowkes *et al.* propose PAM [42] to tackle the problem that the recommended API lists are large and hard to understand. PAM mines API usage patterns through an almost parameter-free probabilistic algorithm and uses them to recommend APIs. Niu *et al.* propose another API usage pattern mining approach, which segments the data using the co-existence relationship of object usages to mine API usage patterns [141]. Liu *et al.* propose RecRank [110] to improve the top-1 accuracy based on API usage paths. Nguyen *et al.* propose FOCUS [139], which mines open-source repositories and analyzes API usages in similar projects to recommend APIs and API usage patterns based on context-aware collaborative-filtering techniques. Wen *et al.* propose FeaRS [222], which mines open-source repositories and extracts API sequences that are implemented together in the same tasks frequently to recommend APIs.

Hindle *et al.* adopt the n-gram model, a widely-used statistical language model, on the code of software [69], and develop a code suggestion tool based on the n-gram model. Tu *et al.* propose to enhance the n-gram model by adding a cache component [203]. Raychev *et al.* propose to extract API sequences

from open-source projects and index them into statistical language models to recommend APIs [177]. Nguyen *et al.* propose a graph-based language model GraLan [136]. Based on GraLan, they design an AST-based language model named ASTLan to recommend APIs. Raychev *et al.* propose a probabilistic model with decision trees named TGEN [175] to predict code tokens. Several recent works try to utilize syntax and data flow information for more accurate recommendation, besides focusing on token sequences [66, 90]. He *et al.* propose PyART [66], which utilizes a predictive model along with data-flow, token similarity and token co-occurrence to recommend APIs. Kim *et al.* leverage Transformer-based techniques to learn the syntactic information from source code [90].

2.3.4 Run-time Environment Conflict Detection

To improve the reliability of software, some researchers work on detecting potential dependency conflicts of software. Artho *et al.* [11] conduct a case study for conflict defects on software packages. Patra *et al.* [150] propose to detect the dependency conflicts between JavaScript libraries. Soto-Valero *et al.* [193] study the problem of multiple versions of the same library co-existing in Maven Central. LibHarmo [75] detects library version inconsistencies for Java Maven projects. Wang *et al.* [212, 213, 214, 215, 216] conduct a series of empirical analyses and develop several tools to facilitate dependency conflict issue diagnosis for the ecosystem of different programming languages.

There are also some research efforts on repairing dependency conflict issues. Su *et al.* [194] propose to repair the inconsistencies between file systems and configuration scripts. Weiss *et al.* [221] capture and replay developer changes to repair the system configuration. HireBuild [64] repairs failing gradle build scripts based on the patterns from TravisTorrent dataset. SmartPip [207] proposes to address the efficiency problem of previous approaches on the PyPI [41] ecosystem.

2.3.5 Run-time Environment Dependency Inference

There are a lot of efforts [75, 212] being devoted to automatically inferring environment dependencies for software. Most recently, DockerizeMe [71] infers third-party and system libraries via static analysis and dynamic analysis. V2 [72] enhances DockerizeMe and explores possible environment dependencies based on feedback-directed search. Pipreqs [156] builds the *requirements.txt* files for Python projects by analyzing the *import* statements in code. SnifferDog [208] builds the execution environments for Python Jupyter notebooks. PyEGo [241] and PyCRE [22] utilize knowledge graphs to represent and analyze the dependencies between the third-party packages used by Python programs.

Chapter 3

Hybrid Type Inference for Python

Type inference has been studied for a long time since the 1970s. Type information provides strong support for the reliability of software. The dynamic type system of Python, however, does not require explicit type declaration in code. This makes it quite necessary to explore type inference in Python to avoid potential type errors that would not occur in static languages. In this chapter, we focus on type inference for Python programs. The main points of this chapter are as follows. (1) we propose a hybrid type inference framework HiTyper that integrates static inference with deep learning for more accurate type prediction. (2) We design an innovative type dependency graph to strictly maintain type dependencies of different variables. (3) We tackle some challenges faced by previous studies and design a series of type rejection rules and a type correction algorithm to validate neural predictions. (4) We conduct extensive experiments to demonstrate the superior performance of HiTyper than state-of-the-art baseline models and static type inference tools in the task.

3.1 Introduction

Dynamically typed programming languages such as Python are becoming increasingly prevalent in recent years. According to GitHub Octoverse 2019 and 2020 [56], Python outranks Java and C/C++ and becomes one of the most popular programming languages. The dynamic features provide more flexible coding styles and enable fast prototyping. However, without concretely defined variable types, dynamically typed programming languages face challenges in ensuring security and compilation performance. According to a recent survey by Jetbrains [82], static typing or at least some strict type hints becomes the top 1 desired feature among Python developers. To address such problems, some research adopts design principles of statically typed programming languages [63, 89, 174]. For example, reusing compiler backend of the statically typed languages [92] and predicting types for most variables [7, 10, 46, 65, 68, 80, 158]. Moreover, Python officially supports type annotations in the Python Enhancement Proposals (PEP) [99, 100, 204, 250].

Type prediction is a popular task performed by existing work. Traditional *static type inference* approaches [10, 46, 65, 80, 176] and type inference tools such as Pytype [166], Pysonar2 [162], and Pyre Infer [160] can correctly infer types for the variables with enough static constraints, e.g., for `a = 1` we can know the type of `a` is `int`, but are unable to handle the variables with few static constraints, e.g. most function arguments or dynamic evaluations such as `eval()` [180].

With the recent development of *deep learning (DL) methods*, we can leverage more type hints such as identifiers and existing type annotations to predict types. Many DL-based methods [7, 68, 114, 129, 158, 235] have been proposed, and they show significant improvement compared with static techniques [95]. While DL-based methods are effective, they face the following two major limitations:

(i) No guarantee of the type correctness. Pradel *et al.* [158] find that the predictions given by DL models are inherently imprecise as they return a list of type candidates for each variable, among which only one type is correct under a certain context. Besides, the predictions made by DL models may contradict the typing rules, leading to type errors. Even the state-of-the-art DL model Typilus [7] generates about 10% of predictions that cannot pass the test of a type checker. The type correctness issue makes the DL-based methods hard to be directly deployed into large codebases without validation. Recent work [7, 158] leverages a search-based validation in which a type checker is used to validate all combinations of types returned by DL models and remove those combinations containing wrong types. However, these approaches cannot correct the wrong types but only filter them out.

(ii) Inaccurate prediction of rare types. Rare types refer to the types with low occurrence frequencies in datasets [7]. Low-frequency problem has become one of the bottlenecks of DL-based methods [88, 108, 173, 178, 243]. For example, Typilus's accuracy drops by more than 50% for the types with occurrence frequencies fewer than 100, compared to the accuracy of the types with occurrence frequencies more than 10,000. More importantly, rare types totally account for a significant amount of annotations even though each of them rarely appears. We analyze the type frequencies of two benchmark datasets from Typilus [7] and Type4Py [127], and find a long tail phenomenon, i.e., the top 10 types in the two datasets already account for 54.8% and 67.8% of the total annotations, and more than 10,000 and 40,000 types in two datasets are rare types with frequency proportions less than 0.1%. They still occupy 35.5% and 25.5% of total annotations for the two respective datasets and become the long "tail" of type distributions.

To remedy the limitations of the previous studies, this chapter proposes a hybrid type inference framework named HiTyper, which conducts static type in-

ference and accepts recommendations from DL models (*Static+DL*). We propose a novel representation, named type dependency graph (TDG), for each function, where TDG records the type dependencies among variables. Based on TDG, we reformulate the type inference task into a blank filling problem where the "blanks" (variables) are connected with dependencies so that both static approaches and DL models can fill the types into "blanks".

HiTyper infers the "blanks" in TDG mainly based on static type inference, which automatically addresses DL models' rare type prediction problem since static type inference rules are insensitive to type occurrence frequencies. HiTyper extends the inference ability of static type inference by accepting recommendations from DL models when it encounters some "blanks" that cannot be statically inferred. Different from the search-based validation by Pradel *et al.* [158], HiTyper builds a series of type rejection rules to filter out all wrong predictions on TDG, and then continues to conduct static type inference based on the reserved correct predictions.

We evaluate HiTyper on two public datasets. One dataset is released by Allamanis *et al.* in the paper of Typilus [7], and the other is ManyTypes4Py [129], one large dataset recently released for this task. Experiment results show that HiTyper outperforms both SOTA DL models and static type inference tools. Compared with two SOTA DL models Typilus and Type4Py, HiTyper presents a 10%∼12% boost on the performance of overall type inference, and a 6% ∼ 71% boost on the performance of certain kinds of type inference such as return value type inference and user-defined type inference. Without the recommendations from neural networks and only looking at the static type inference part, HiTyper generally outputs 2× ∼ 3× more annotations with higher precision than current static type inference tools Pyre [160] and Pytype [166]. HiTyper can also identify wrong human annotations in real-world projects. We identify seven

wrong annotations in six projects of Typilus's dataset and submit pull requests to correct these annotations. Two project owners have approved our corrections.

Contributions. Our contributions can be concluded as follows:

- To the best of our knowledge, we are the first to propose a hybrid type inference framework that integrates static inference with DL for more accurate type prediction.
- We design an innovative type dependency graph to strictly maintain type dependencies of different variables.
- We tackle some challenges faced by previous studies and design a series of type rejection rules and a type correction algorithm to validate neural predictions.
- Extensive experiments demonstrate the superior performance of the proposed HiTyper than SOTA baseline models and static type inference tools in the task.

3.2 Motivation

Listing 3.1 illustrates an example of code snippet from the WebDNN project.[1] Results of several baselines, including static type inference techniques - Pytype and Pysonar2, and state-of-the-art DL models - Typilus, are depicted in Table 3.1.

```
#src/graph_transpiler/webdnn/graph/shape.py
def parse(text):
    normalized_text = _normalize_text(text)
    tmp = ast.literal_eval(normalized_text)
    shape = []
```

[1] https://github.com/mil-tokyo/webdnn

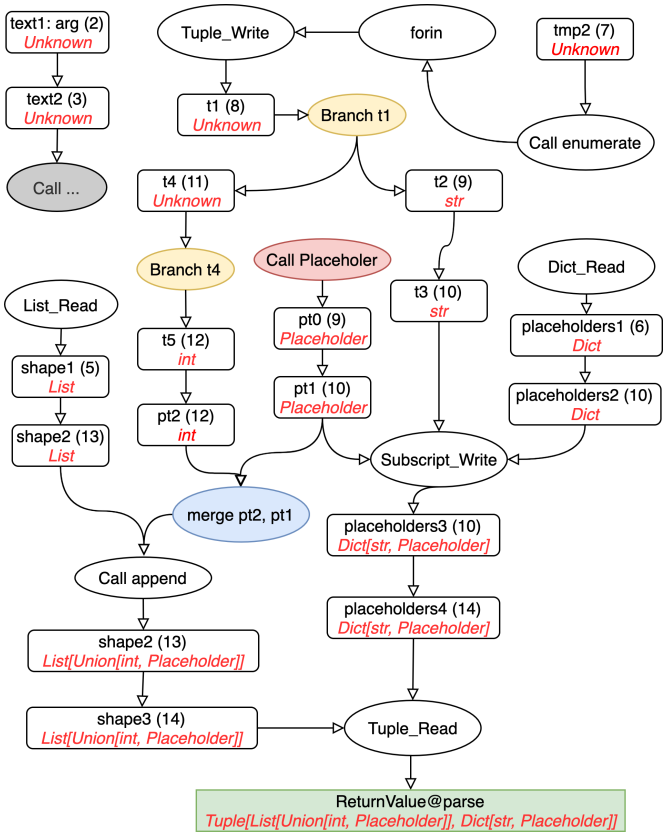


Figure 3.1: Type dependency graph of the parse() from Code.3.1.

```
    placeholders = {}
    for i, t in enumerate(tmp):
        if isinstance(t, str):
            pt = Placeholder(label=t)
            placeholders[t] = pt
        elif isinstance(t, int):
            pt = t
        shape.append(pt)
    return shape, placeholders
```

Listing 3.1: A Function from WebDNN.

Static Inference. According to Table 3.1, we can find that the static type

Table 3.1: Prediction results of different baselines for Listing 3.1.

Approach	Baseline	Argument	Return Value
	Ground Truth	str	Tuple[List[int, Placeholder], Dict[str, Placeholder]]
Static	Pysonar2	?	Tuple[List[int],Dict]
	Pytype	?	Tuple[List, Dict]
DL	Typilus	1. str	1. Tuple[collections.OrderedDict[Text, List[DFAState]], Optional[Text]],Tuple[Any, List[Tuple[Any]], Any] 2. Tuple[Text] 3. Tuple[torch.Tensor]
Static + DL	HiTyper (Typilus)	str	Tuple[List[int, Placeholder], Dict[str, Placeholder]]

inference techniques fail to infer the type of the argument `text` since the argument is at the beginning of data flow without any assignments or definitions. One common solution to infer the type is to use inter-procedural analysis and capture the functions that call `parse()` [186]. However, tracing the functions in programs, especially in some libraries, is not always feasible. As for the return value, by analyzing the data flow and dependencies between variables, static inference can easily identify that `shape` (line 5, 13) and `placeholders` (line 6, 10) consist of the return value. It can recursively analyze the types of the two variables and finally output the accurate type of the return value. Indeed, both Pysonar2 and Pytype can correctly infer that the return value is a tuple containing a list and dict.

DL Approach. The DL model Typilus [7] accurately predicts the type as `str` according to the semantics delivered by the argument `text` and contextual

information. The case illustrates that DL models can predict more types than static inference. However, Typilus fails to infer the type of the return value of `parse()`. Current DL models cannot maintain strict type dependencies between variables. Therefore, Typilus only infers the type as a tuple but cannot accurately predict the types inside the tuple. When adding a type checker to validate Typilus's predictions, its argument prediction is reserved since it does not violate any existing type inference rules. However, for the return value, its 2nd and 3rd type predictions in Table 3.1 by Typilus are rejected since the return value of `parse()` explicitly contains two elements with different types. The 1st prediction is also rejected because it contains the type `Optional[text]` that does not appear in the return value. In this case, the model does not produce any candidate type for the return value.

Static+DL Approach. For the code example, we find that static inference is superior to DL models when sufficient static constraints or dependencies are satisfied, while DL models are more applicable for the types lacking sufficient static constraints. Given the code, HiTyper first generates the TDG of it, as shown in Fig. 3.1, and tries to fill all nodes in TDG with corresponding types ("blank filling"). For the argument `text`, HiTyper identifies that the type cannot be inferred by static inference (it does not have any input edges) and asks DL for recommendations. HiTyper does not directly output the predictions from DL as final type assignments. Instead, HiTyper validates the prediction's correctness and accepts the result only if no type inference rules are violated. When predicting the return value, HiTyper captures its type dependencies based on the TDG (it connects with two input nodes) and directly leverages static inference to infer the type. For this case, DL predictions are not required, largely avoiding the imports of wrong types.

3.3 HiTyper

In this section, we first introduce the definitions used in HiTyper and then elaborate on the details of HiTyper.

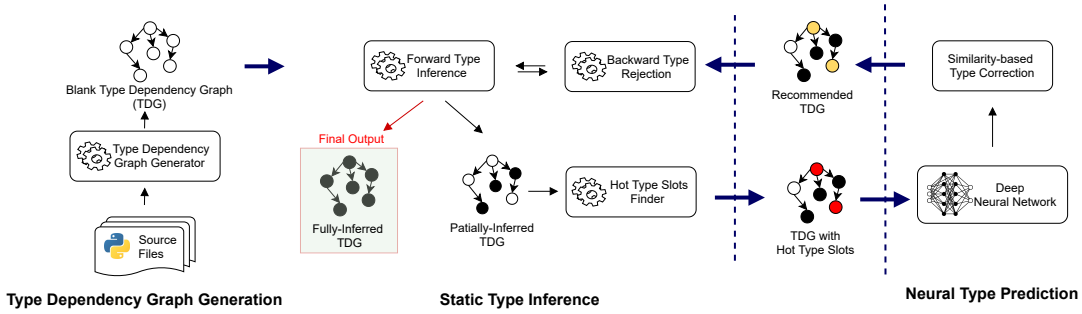


Figure 3.2: Overall architecture of HiTyper. Black solid nodes, hollow nodes, red nodes, and yellow nodes in the type dependency graphs represent inferred type slots, blank type slots, hot type slots, and the type slots recommended by the DL model, respectively.

3.3.1 Overview

HiTyper accepts Python source files as input and outputs JSON files recording the type assignment results. Fig. 3.2 illustrates its overall architecture. HiTyper includes three major components: type dependency graph generation, static type inference, and neural type prediction. The static type inference component comprises two main steps, i.e., forward type inference and backward type rejection.

Type Dependency Graph (TDG) Generation. Specifically, given a Python source file, HiTyper first generates TDGs for each function and identifies all the imported user-defined types (Sec. 3.3.2). TDG transforms every variable occurrence and expression into nodes and maintains type dependencies between

them so that static inference and DL models can work together to fill types into it.

Static Type Inference - Forward Type Inference. To maintain the correctness of prediction results, HiTyper focuses on inferring types using static inference. Given a TDG, HiTyper conducts forward type inference by walking through the graph and implementing the type inference rules saved in each expression node (Sec. 3.3.3). However, due to the limitation of static inference, in most cases HiTyper can only infer partial type slots, i.e., variables, indicated as black solid nodes in the *partially-inferred TDG* in Fig. 3.2; while the blank nodes denote the type slots without sufficient static constraints and remaining unsolved. To strengthen the inference ability of HiTyper, we ask DL models for recommendations.

Neural Type Recommendation. Through the hot type slot finder, HiTyper identifies a key subset of the blank nodes as hot type slots, marked as red nodes in Fig. 3.2, for obtaining recommendations from DL models. HiTyper also employs a similarity-based type correction algorithm to supplement the prediction of user-defined types, which are the primary source of rare types (Sec. 3.3.4). The types recommended by the neural type prediction component are filled into the graph, resulting in the *recommended TDG*.

Static Type Inference - Backward Type Rejection. HiTyper utilizes type rejection rules to validate the neural predictions in hot type slots (Sec. 3.3.3). Then, it traverses the whole TDG to transmit the rejected predictions from output nodes to input nodes so that all nodes in TDG can be validated. Finally, HiTyper invokes forward type inference again to infer new types based on the validated recommendations.

The interactions between forward type inference and backward type rejection could iterate until the TDG reaches a fixed point, i.e., the types of all nodes do not

change anymore. Meanwhile, the iterations between static inference and neural prediction can repeat several times until all type slots are inferred, or a maximum iteration limit is reached.

3.3.2 Type Dependency Graph Generation

This section introduces the creation of the type dependency graph (TDG), which describes the type dependencies between different variables in programs. Fig. 2.2 presents the syntax of all the expressions that generate types in Python, where each expression corresponds to a node in the AST (Abstract Syntax Tree). Given the AST of a program, HiTyper can quickly identify these expressions. The expression nodes constitute a major part of TDG. We define TDG as below.

Definition. We define a graph $G = (N, E)$ as a type dependency graph (TDG), where $N = \{n_i\}$ is a set of nodes representing all variables and expressions in source code, and E is a set of directed edges of $n_i \rightarrow n_j$ indicating the type of n_j can be solved based on the type of n_i by type inference rules. We also denote n_i is the input node of n_j and n_j is the output node of n_i here.

The TDG contains four kinds of nodes:

- *symbol* nodes represent all the variables for which the types need to be inferred. We also use *type slots* to indicate symbol nodes in the following sections.
- *expression* nodes represent all the expressions that generate types as shown in Fig. 2.2.
- *branch* nodes represent the branch of data flows.
- *merge* nodes represent the merge of data flows.

HiTyper creates a node for every variable occurrence instead of every variable in TDG because Python's type system allows variables to change their types at run-time. Similar to static single assignment (SSA), HiTyper labels each occurrence of a variable with the order of occurrences, so that each symbol node in the TDG has a format of `$name$order($lineno)` to uniquely indicate a variable occurrence. For example, in Fig. 3.1, we create three symbol nodes (`pt0(9)`, `pt1(10), pt2(12)`) for variable `pt` as it appears three times in Listing 3.1 (Line 9, 10, and 12).

Algorithm 1 Type Dependency Graph Generation

Input: The AST of given function, *func_ast*;

Output: Type dependency graph of the given function, *tg*

Initialize an expression stack *ex_stack*
Initialize a variable stack *var_stack*

for all *node* ∈ func_ast && *node* is not visited **do**
    // handle expression nodes
    if *node*.type ∈ Expressions **then**
        ex_stack.push(*node*); *ex_node* ← new ex(*node*)
        visit(*node*.operands); *ex_stack*.pop(*node*)
        if not *ex_stack*.empty() **then**
            tg.addEdge(*ex_node* → *ex_stack*.top())
        end if
        tg.addNode(*ex_node*)
    end if
    // handle symbol nodes
    if *node*.type == ast.Name **then**

```
17:         sym_node ← new symbol(node)
18:         if node.ctx == write then
19:             tg.addEdge(ex_stack.top() → sym_node)
20:         else
21:             tg.addEdge(var_stack.top() → sym_node)
22:             tg.addEdge(sym_node → ex_stack.top())
23:         end if
24:         var_stack.push(sym_node); tg.addNode(sym_node)
25:     end if
26:     // handle branch and merge nodes
27:     if checkTypeBranch(node) then
28:         branch_node ← new branch(node)
29:         tg.addNode(branch_node)
30:         ctx1, ctx2 ← Branch(ctx)
31:         visit(node.left, ctx1); visit(node.right, ctx2)
32:     end if
33:     if checkTypeMerge(node) then
34:         merge_node ← new merge(node)
35:         tg.addNode(merge_node)
36:         ctx ← Merge(ctx1, ctx2)
37:     end if
38: end for
```

Import Analysis. Before establishing TDG for every input function, HiTyper first conducts import analysis to extract all user-defined types so that it can distinguish the initialization of user-defined types from regular function calls. HiTyper first collects all classes in source files, which constitute the initial set of user-defined types. Then it analyzes all local import statements such as "*from*

package import class", and adds the imported classes into the user-defined type set. For all global import statements such as "*import package*", HiTyper locates the source of this package and adds all the classes and named tuples in the source into the user-defined type set. For each imported class, HiTyper solves the location of external source files and checks whether operator overloading methods exist in this class.

Type Dependency Graph Generation. Given the AST of input code and all the user-defined types extracted by import analysis, HiTyper creates TDG for each function based on the main logic shown in Alg. 1. HiTyper first locates all the variables and expressions in the code by traversing the whole AST. Specifically, to visit each AST node, HiTyper employs the ASTVisitor provided by Python's module *ast* [163]. HiTyper identifies expressions according to the definitions of expression nodes in Python (as depicted in Fig. 2.2) and records every visited expression node using an expression stack. Whenever HiTyper identifies an expression node (Line 3), it builds the same node in the current TDG and pushes it into the expression stack. HiTyper will then recursively visit the expression's operands to capture new expression nodes until it encounters a variable node (Line 12), which is the leaf node of the AST.

HiTyper builds a symbol node in TDG for each visited identifier node of AST and maintains a variable map to record all the occurrences of each variable. The AST already indicates the context of each variable occurrence, i.e., whether *read* or *write*.

(i) If the variable context is *read*, HiTyper will obtain the last occurrence of the variable according to the maintained variable map under the current context. It then creates an edge from the symbol node of the last occurrence to the symbol node of the current variable (Line 16 - 18).

(ii) If the variable context is *write*, HiTyper will fetch the value from the

last expression in the expression stack and build an edge connecting from the expression node to the symbol node of the current variable (Line 14 - 15).

Analogous to regular data flow analysis, HiTyper also checks whether the data flow branches (Line 23 - 27) or merges at certain locations (Line 29 - 32).

In TDG, each symbol node keeps a list of candidate types, while each expression node includes type inference rules and type rejection rules. When HiTyper walks through TDG, the rules will be activated to produce new types. Thus, types can *flow* from arguments to return values. By traversal, HiTyper obtains the types of each symbol node and outputs the type assignment. The leveraged type inference rules and type rejection rules are detailed in the next subsections.

3.3.3 Static Type Inference

This section describes the type inference and rejection rules integrated into expression nodes, which are the key components of our static type inference. Fig. 3.3 and Fig. 3.4 denote all the type inference and rejection rules used in static type inference. Each rule consists of some premises (contents above the line) and conclusions (contents below the line). They obey the following form:

$$\pi \vdash e : \theta.$$

In this form, π is called the context, which includes lists that assign types to expression patterns. e is the expression showed in Fig. 2.2, and we use $e_1, ..., e_n$ to represent different expressions. θ is the type showed in Fig. 2.1. We use $\theta_1, ..., \theta_n$ to represent different types. A rule under this form is called a *type judgment* or *type assignment*. Our goal is to get the context π that assigns types to all the variables in the code.

The premises of each rule in Fig. 3.3 and Fig. 3.4 are the types of input nodes $\theta_1, \theta_2, ...$ that construct an expression, and the valid type set $\widetilde{\theta}$ for the current operation. Usually, type inference rules only have one conclusion, which

$$\frac{v \in \mathbf{Dom}(\pi)}{\pi \vdash v : \theta} \quad \text{(Variable)}$$

$$\frac{}{\pi \vdash c : \theta} \quad \text{(Constant)}$$

$$\frac{\pi \vdash e : \theta}{\pi \vdash e.v : \theta.v} \quad \text{(Attribute)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\mathbf{bool},\ \mathbf{int},\ O\}}{\pi \vdash e_1\ \mathbf{bitop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \quad \text{(LShift, RShift)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \widetilde{\theta} = \{\mathbf{bool},\ \mathbf{int},\ \mathbf{float},\ O\} \quad \pi \vdash e_2 : \theta_2 \quad \theta' = getMorePreciseType(\theta_1 \wedge \widetilde{\theta}, \theta_2 \wedge \widetilde{\theta})}{\pi \vdash e_1\ \mathbf{numop}\ e_2 : \theta' \quad \pi \vdash \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash \theta_2 \wedge \widetilde{\theta}} \quad \text{(Numeric Operations)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \widetilde{\theta_1} = \{\mathbf{int}, \mathbf{bool}\} \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta_2} = \{\Gamma,\ \mathbf{List},\ \mathbf{Tuple},\ O\}}{\pi \vdash e_1\ \mathbf{numop}\ e_2 : \theta_2 \wedge \widetilde{\theta_2} \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta_1} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta_2}} \quad \text{(Mult)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\Gamma,\ \mathbf{List},\ \mathbf{Tuple},\ O\}}{\pi \vdash e_1\ \mathbf{cmpop}\ e_2 : \mathbf{bool} \quad \pi \vdash e_1 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}} \quad \text{(Lt,LtE,Gt,GtE)}$$

$$\frac{}{\pi \vdash u(e_1, ..., e_n) : u} \quad \text{(Class Instantiation)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad ... \quad \pi \vdash e_n : \theta_n}{\pi \vdash (e_1, ..., e_n) : \mathbf{Tuple}[\theta_1, ..., \theta_n] \quad \pi \vdash [e_1, ..., e_n] : \mathbf{List}[\theta_1, ..., \theta_n]}$$

$$\pi \vdash \{e_1, ..., e_n\} : \mathbf{Set}[\theta_1, ..., \theta_n] \quad \text{(Tuple, List, Set)}$$

$$\frac{\pi \vdash e : \theta \quad \widetilde{\theta} = \{A, \mathbf{str}, \mathbf{bytes}\} \quad \theta' = getElementType(\theta \wedge \widetilde{\theta})}{\pi \vdash \mathbf{for}\ \ v\ \mathbf{in}\ e : \theta' \quad \pi \vdash e : \theta \wedge \widetilde{\theta}} \quad \text{(Comprehension)}$$

$$\frac{\pi \vdash \mathbf{for}\ v\ \mathbf{in}\ e_1 : \theta_1 \quad \pi \vdash e_2[v] : \theta_2}{\pi \vdash (e_2[v]\ \mathbf{for}\ v\ \mathbf{in}\ e_1) : \mathbf{Generator}[\theta_2]} \quad \text{(Generator)}$$

$$\frac{\pi \vdash \mathbf{for}\ v\ \mathbf{in}\ e_1 : \theta_1 \quad \pi \vdash e_2[v] : \theta_2}{\pi \vdash [e_2[v]\ \mathbf{for}\ v\ \mathbf{in}\ e_1] : \mathbf{List}[\theta_2] \qquad \pi \vdash \{e_2[v]\ \mathbf{for}\ v\ \mathbf{in}\ e_1\} : \mathbf{Set}[\theta_2]} \quad \text{(List, Set Comprehension)}$$

Figure 3.3: Type inference and rejection rules of expressions in Python - Part I

is the result type of the current expression. However, as we also involve neural predictions in TDG and use type rejection rules to validate them, the conclusions of each rule in Fig. 3.3 and Fig. 3.4 have two parts: 1) the result type of the current expression node and 2) the validated types of input nodes. (Some rules may not have the second part because they accept any input types.)

The result type of the current expression node is what traditional static type inference techniques usually infer. We denote it as *forward type inference*. However, there exist types that are not allowed to conduct certain operations, which are guided by *type constraints*. When a type constraint is violated, e.g.,

adding an integer to a string, traditional static inference techniques [133, 160, 166] throw type errors. For the wrongly predicted cases, HiTyper does not directly throw a type error since it accepts recommendations from DL models. To "sanitize" the recommendations from DL models, we create type rejection rules to validate and remove the wrong predictions in input nodes. We call this as *backward type rejection*.

Forward Type Inference. HiTyper starts forward type inference with the nodes that do not have input nodes in TDG. It gradually visits all nodes in the graph and activates corresponding type inference rules if their premises are satisfied, i.e., all input nodes are fully inferred. This is the forward traversal of TDGs. As forward type inference in HiTyper is similar to traditional static type inference techniques, we only discuss the [Call] rule for which HiTyper has a special strategy. The premise of the [Call] rule requires the type of callees, which is beyond the scope of current functions. This premise is one major barrier for most static inference techniques to fully infer a program due to a large number of external APIs in Python programs [73, 186]. HiTyper only focuses on inferring the types of functions with explicit implementation in the current source code, in which the TDGs of the functions are connected. HiTyper does not infer external calls for two reasons: 1) DL models perform well on predicting the types of commonly-used APIs that frequently occur in the training set; 2) Python maintains a *typeshed* [165] project to collect the type annotations of frequently-used modules, so HiTyper can directly access the types.

Backward Type Rejection. An input type in an expression must fulfill two constraints before it can conduct the expression: 1) it must be the valid type to conduct a certain expression, 2) it must have a valid relationship with other input types. HiTyper rejects the input types that violate these two constraints. It first checks whether the type is valid for an expression. We indicate valid types

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2}{\pi \vdash e_1 \ \textbf{blop}\ e_2 : \textbf{Union}[\theta_1, \theta_2]} \quad \text{(Boolean Operations)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\textbf{bool},\ \textbf{int},\ \textbf{Set},\ O\}}{\pi \vdash e_1 \ \textbf{bitop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \quad \text{(BitOr, BitAnd, BitXor)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\Gamma,\ \textbf{List},\ \textbf{Tuple},\ O\}}{\pi \vdash e_1 \ \textbf{numop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}} \quad \text{(Add)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\Gamma,\ \textbf{Set},\ O\}}{\pi \vdash e_1 \ \textbf{numop}\ e_2 : \theta \wedge \widetilde{\theta} \quad \pi \vdash e_1 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta} \quad \pi \vdash e_2 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}} \quad \text{(Sub)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2}{\pi \vdash e_1 \ \textbf{cmpop}\ e_2 : \textbf{bool}} \quad \text{(Eq,NotEq,Is,IsNot)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta} = \{\textbf{str}, \textbf{bytes}, \textbf{List},\ \textbf{Tuple},\ \textbf{Set},\ \textbf{Dict},\ \textbf{Generator}\}}{\pi \vdash e_1 \ \textbf{cmpop}\ e_2 : \textbf{bool} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \quad \text{(In,NotIn)}$$

$$\frac{\pi \vdash e : \theta \quad \pi \vdash e_1 : \theta_1 \quad ... \quad \pi \vdash e_n : \theta_n \quad \widetilde{\theta} = \{\textbf{Callable}[[\theta_1, ..., \theta_n], \theta]\} \quad \theta' = getReturnType(\theta \wedge \widetilde{\theta})}{\pi \vdash e(e_1, ..., e_n) : \theta} \quad \text{(Call)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad ... \quad \pi \vdash e_n : \theta_n}{\pi \vdash \{e_1 : e_2, ..., e_{n-1} : e_n\} : \textbf{Dict}[\theta_1 : \theta_2, ..., \theta_{n-1} : \theta_n]} \quad \text{(Dict)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta_1} = \{\textbf{Dict}\} \quad \theta' = getValueType(\theta_1 \wedge \widetilde{\theta_1})}{\pi \vdash e_1[e_2] : \theta' \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta_1}} \quad \text{(SubScript)}$$

$$\frac{\pi \vdash \textbf{for}\ v\ \textbf{in}\ e_1 : \theta_1 \quad \pi \vdash e_2[v] : \theta_2 \quad \pi \vdash e_3[v] : \theta_3}{\pi \vdash \{e_2[v] : e_3[v]\ \textbf{for}\ v\ \textbf{in}\ e_1\} : \textbf{Dict}[\theta_2 : \theta_3]} \quad \text{(Dict Comprehension)}$$

$$\frac{\pi \vdash e_1 : \theta_1 \quad \pi \vdash e_2 : \theta_2 \quad \widetilde{\theta_1} = \{A, \textbf{str}, \textbf{bytes}\} \quad \widetilde{\theta_2} = \{\textbf{int}, \textbf{bool}\} \quad \theta' = getElementType(\theta_1 \wedge \theta_2)}{\pi \vdash e_1[e_2] : \theta' \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta_1} \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta_2}} \quad \text{(Slice)}$$

Figure 3.4: Type inference and rejection rules of expressions in Python - Part II

for each expression as $\widetilde{\theta}$ in Fig. 3.3 and Fig. 3.4. For example, in [In, NotIn] rule, the types of e_2 must be iterable so `int` is not allowed and should not be in the valid type set $\widetilde{\theta}$. Then HiTyper checks whether the relationships between all inputs are valid. Apart from valid types for a certain operation, some operations also require the inputs to satisfy a certain relationship. For example, in [Add] rule, the two operands must have the same type. For types of two inputs `int` and `str`, even though they are in the valid type set of this operation, they are still rejected because they are not the same type. Therefore, in the [Add] rule, the

final valid input types are the intersection of all input type sets θ_1, θ_2 and valid type set $\widetilde{\theta}$.

Type Rejection rules can validate and reject the input types of an operation. However, the input types are the results of previous operations, so the type rejection process will also affect the input types of previous operations. To thoroughly remove the influence of wrong types, HiTyper also rejects the input types that result in the rejected types according to forward type inference rules. HiTyper gradually validates all type slots by starting from the type slots without output edges and producing the rejected input types. Then, it traverses other slots until the whole TDG is visited. This is the backward traversal of TDGs.

Correctness. Different from the DL-based approaches [68, 129], HiTyper can always guarantee the correctness of its type assignments based on static inference. According to the architecture of HiTyper in Fig. 3.2, the type assignments generated by HiTyper have two cases: 1) If the static inference can successfully handle a program, HiTyper does not need to invoke DL models to give type recommendations. Consequently, the type assignments fully based on the inference rules (Fig. 3.3 and Fig. 3.4) are sound because they are collected from the Python official implementation CPython [38]; and 2) If the static inference cannot fully infer a program and the DL models are invoked to provide type recommendations (Sec. 3.3.4), HiTyper utilizes type rejection rules to validate the recommendations and then calls the type inference rules again to infer the remaining types. In this case, our rejection rules thoroughly eliminate the influence of wrong recommendations, and the final results are also produced by static inference. Therefore, HiTyper always maintains the type correctness.

3.3.4 Neural Type Recommendation

HiTyper conducts static type inference based on type inference rules. When static type inference can fully infer all the variables in TDG. However, some variables are hard to be statically typed so that HiTyper only gets a partially-inferred TDG. In this case, HiTyper asks DL models for recommendations. The neural type recommendation part of HiTyper includes two procedures: hot type slot identification and similarity-based type correction.

Hot Type Slot Identification. Some variables can impact the types of many other variables because they locate at the beginning of the data flow or possess type dependencies with many variables. We call these variables as *hot type slots*. Given the types of hot type slots, static type inference techniques can infer the remaining type slots. Therefore, to optimize the type correctness of HiTyper, DL models are only invoked on the hot type slots instead of all the blank type slots.

To identify the hot type slots, HiTyper first removes slots already filled by static type inference and obtains a sub-graph with all the blank type slots. Then HiTyper employs a commonly-used dominator identification algorithm semi-NCA [54] to capture all dominators in the sub-graph. A node X dominating another node Y in a graph means that each entry node to Y must pass X. Thus, if a type slot X dominates another type slot Y, Y's type can be inferred from X's type. HiTyper gradually removes the type slots Y from the sub-graph until no type slots can be removed. In the smallest sub-graph, each type slot is not dominated by other type slots, and all the slots are hot type slots. For these type slots, HiTyper accepts type recommendations from DL models.

Similarity-based Type Correction for User-defined Types. DL models provide one or more type recommendations for each hot type slot, depending on the strategy (Top-1, -3, or -5) HiTyper uses. Some DL models [68, 158]

treat user-defined types as OOV tokens and do not predict the types, while other models [7, 166] directly copy user-defined types from the training set but fail to predict those never appearing in the training set. We propose to complement the recommendation of user-defined types using the similarity-based type correction algorithm shown in Alg. 2. Note that HiTyper only focuses on replacing the explicitly incorrect user-defined types, i.e., those never imported or defined in the current source file, with the most similar user-defined types collected by import analysis.

Algorithm 2 Type correction of user-defined types

Input: Variable name, $name$;
Valid user defined type set, S;
Type String recommended by deep neural networks, t;
Penalty added for name-type similarity to align with type-type similarity, $penalty$;
Output: Corrected type of current variable, ct;
if $t \in S$ or isBuiltin(t) **then**
    $ct \leftarrow t$;
else
    largest_sim $\leftarrow 0$; largest_type $\leftarrow None$;
    $tw \leftarrow$ BPE(t); $namew \leftarrow$ BPE($name$);
    for each $pt \in S$ **do**
        $ptw \leftarrow$ BPE(pt);
        if $sim(ptw, tw) >$ largest_sim **then**
            largest_sim $\leftarrow$ sim(ptw, tw); largest_type $\leftarrow pt$;
        end if
        if $sim(ptw, namew) + penalty >$ largest_sim **then**
            largest_sim $\leftarrow$ sim($ptw, namew$); largest_type $\leftarrow pt$;

16: **end if**

17: **end for**

18: $ct \leftarrow$ largest_type;

19: **end if**

Specifically, if the recommended type does not belong to built-in types, HiTyper checks whether the type appears in the user-defined type set collected from import analysis (Line 1). If the check result is False, the type will be regarded as explicitly incorrect and should be corrected. For these incorrect user-defined types, HiTyper replaces them with the most similar candidate in the user-defined type set. HiTyper employs Word2Vec [123] to embed two types and the variable name into word embeddings and calculates the cosine distance as the similarity of the two types (Line 6-12). For the OOV tokens, HiTyper splits them into subtokens using the BPE algorithm [47, 189] (Line 5). Finally, HiTyper chooses the type candidate with the largest similarity to fill the user-defined type (Line 15).

3.4 Evaluation

In the section, we answer the following research questions:

RQ1: How effective is HiTyper compared to baseline approaches?

RQ2: Can HiTyper well predict the rare types?

RQ3: What is the performance of the static type inference component in HiTyper?

Table 3.2: Type distribution in the test set. "Rare" indicates rare types and "User" indicates user-defined types.

	Category	Total	Rare	User	Arg	Return	Local
Typilus	**Count**	15,772	7,103	5,572	11,261	4,511	-
	Prop.	100%	45%	35%	71%	29%	-
Type4Py	**Count**	37,408	14,035	10,023	11,807	5,491	20,110
	Prop.	100%	37%	27%	32%	15%	53%

3.4.1 Experimental Setup

Dataset. We used the two Python datasets mentioned in Sec. 3.2 for evaluation. One is the *Typilus's Dataset* released by Allamanis *et al.* [7]; and the other one is *ManyTypes4Py* released by Mir *et al.* [127], with the number of different types in the test set and more detailed statistics shown in Table 3.2 and Sec. 3.2, respectively.

Evaluation Metrics. Following the previous work [7, 129], we choose two metrics *Exact Match* and *Match to Parametric* for evaluation. The two metrics compute the ratio of results that: 1) *Exact Match*: completely matches human annotations. 2) *Match to Parametric*: satisfy exact match when ignoring all the type parameters. For example, `List[int]` and `List[str]` are considered as matched under this metric.

Baseline Approaches. To verify the effectiveness of the proposed HiTyper, we choose five baseline approaches for comparison:

1) A naive baseline. It represents a basic data-driven method. We build this baseline following the work [158], which makes predictions by sampling form the distribution of the most frequent ten types.

2) Pytype [166] and Pyre Infer [160]. They are two popular Python static type inference tools from Google and Facebook, respectively.

3) Typilus [7] and Type4Py [129]. Typilus is a graph model that utilizes code structural information. Type4Py is a hierarchical neural network that uses type clusters to predict types.

Implementation of HiTyper The entire framework of HiTyper is implemented using Python, which contains more than 9,000 lines of code. We obtain all typing rules and rejection rules from Python's official documentation [37] and its implementation CPython[2]. We use the Word2Vec model from the gensim library [251] as the embedding when calculating the similarity between the two types. We train the Word2Vec model by utilizing all the class names and variable names in the training set of Typilus. The dimensions of the word embeddings and the size of the context window are set as 256 and 10, respectively. Due to the small training corpus for Word2Vec, we choose the Skip-Gram algorithm for model training [124]. We choose Typilus and Type4Py as the neural network model from which HiTyper accepts type recommendations. We chose the exact hyper-parameters for Typilus, and Type4Py used in the original papers. We run all experiments on Ubuntu 18.04. The system has an Intel(R) Xeon(R) CPU (@2.4GHz) with 32GB RAM and 2 NVIDIA Titan V GPUs with 12GB RAM.

3.4.2 RQ 1: Effectiveness of HiTyper

We evaluate the effectiveness of HiTyper considering different type categories, including arguments, local variables, and return values. The results are depicted in Table 3.3.

Overall performance. The naive baseline achieves high scores regarding the top-5 exact match metric for different type categories, some of which are even close to the performance of DL models. Since the naive baseline only predicts types with high occurrence frequencies in the dataset, the results indicate

[2]https://github.com/python/cpython

Table 3.3: Comparison with the baseline approaches. Top-1,3,5 of HiTyper means it accepts 1,3,5 candidates from deep neural networks in the type recommendation phase. The neural network in HiTyper is the corresponding comparison DL model.

Dataset	Type Category	Approach	Top-1		Top-3		Top-5	
			Exact Match	Match to Parametric	Exact Match	Match to Parametric	Exact Match	Match to Parametric
ManyTypes4Py	Argument	Naive Baseline	0.14	0.16	0.33	0.38	0.43	0.51
		Type4Py	0.61	0.62	0.64	0.66	0.65	0.68
		HiTyper	**0.65**	**0.67**	**0.70**	**0.74**	**0.72**	**0.76**
	Return Value	Naive Baseline	0.07	0.10	0.19	0.28	0.28	0.42
		Type4Py	0.49	0.52	0.53	0.59	0.54	0.63
		HiTyper	**0.60**	**0.72**	**0.63**	**0.76**	**0.65**	**0.77**
	Local Variable	Naive Baseline	0.13	0.17	0.33	0.45	0.47	0.65
		Type4Py	0.67	0.73	0.71	0.78	0.72	0.79
		HiTyper	**0.73**	**0.85**	**0.74**	**0.86**	**0.75**	**0.86**
	All	Naive Baseline	0.13	0.16	0.31	0.40	0.43	0.57
		Type4Py	0.62	0.66	0.66	0.72	0.67	0.73
		HiTyper	**0.69**	**0.77**	**0.72**	**0.81**	**0.72**	**0.82**
Typilus's Dataset	Argument	Naive Baseline	0.19	0.20	0.38	0.42	0.46	0.50
		Typilus	0.60	0.65	0.69	0.74	0.71	0.76
		HiTyper	**0.63**	**0.68**	**0.72**	**0.76**	**0.76**	**0.79**
	Return Value	Naive Baseline	0.11	0.11	0.28	0.31	0.36	0.43
		Typilus	0.41	0.57	0.48	0.62	0.50	0.64
		HiTyper	**0.57**	**0.70**	**0.63**	**0.75**	**0.64**	**0.77**
	All	Naive Baseline	0.17	0.18	0.35	0.39	0.44	0.48
		Typilus	0.54	0.62	0.63	0.70	0.65	0.72
		HiTyper	**0.61**	**0.69**	**0.69**	**0.76**	**0.72**	**0.78**

the challenge of accurately predicting rare types. Typilus and Type4Py mitigate the challenge by using similarity learning and type clusters and achieve ∼0.6 regarding the top-1 exact match metric. HiTyper further improves the metric by 11% and 15% compared with Typilus and Type4Py, respectively. HiTyper also enhances the top-1 match to parametric metric by 17% and 11% compared with Typilus and Type4Py, respectively. The improvement indicates the effectiveness of HiTyper in accurate type prediction. Besides, HiTyper presents better performance than the respective DL models regarding the top-3,5 metrics,

demonstrating that HiTyper infers new results based on the static type inference rules, instead of just filtering out or reordering the predictions of DL models.

Type categories. Both Type4Py and Typilus perform better on the argument category than the return value category, which may reflect the difficulty of predicting the types of return values. By building upon type inference rules and TDGs, HiTyper can handle the complicated type dependencies of return values and thereby improve Type4Py and Typilus by 22% and 39%, respectively, w.r.t. the Top-1 exact match metric. HiTyper also slightly meliorates the prediction of the argument category by 7% and 5% compared with Type4Py and Typilus, respectively. The improvement may be attributed to the type correction for user-defined types. Moreover, HiTyper outperforms Type4Py by 9% for predicting local variables.

Answer to RQ1: HiTyper shows great improvement (11% $\sim$ 15%) on overall type inference performance, and the most significant improvement is on return value inference (22% $\sim$ 39%).

3.4.3 RQ 2: Prediction of Rare Types

Rare types are defined as the types with proportions less than 0.1% among the annotations in the datasets, and we observe that 99.7% and 79.0% of rare types are user-defined types in ManyTypes4Py and Typilus's dataset, respectively. Table 3.4 illustrates the prediction results of rare types and user-defined types. We can observe that the naive baseline barely infers rare types and user-defined types. Besides, the performance of Type4Py and Typilus drops significantly for the two type categories, which indicates that type occurrence frequencies can impact the performance of DL models. HiTyper shows the best performance on predicting the two type categories. Specifically, for inferring the

Table 3.4: Comparison with the baseline DL approaches.

Dataset	Type Category	Approach	Top-1		Top-3		Top-5	
			Exact Match	Match to Parametric	Exact Match	Match to Parametric	Exact Match	Match to Parametric
ManyTypes4Py	User-defined Types	Naive Baseline	0.00	0.00	0.00	0.00	0.00	0.00
		Type4Py	0.29	0.29	0.34	0.34	0.36	0.36
		HiTyper	**0.49**	**0.49**	**0.56**	**0.56**	**0.58**	**0.58**
	Rare Types	Naive Baseline	0.03	0.07	0.08	0.21	0.13	0.35
		Type4Py	0.39	0.46	0.45	0.54	0.47	0.57
		HiTyper	**0.51**	**0.66**	**0.56**	**0.72**	**0.58**	**0.73**
Typilus's Dataset	User-defined Types	Naive Baseline	0.00	0.00	0.00	0.00	0.00	0.00
		Typilus	0.32	0.32	0.40	0.40	0.42	0.42
		HiTyper	**0.47**	**0.47**	**0.56**	**0.56**	**0.60**	**0.60**
	Rare Types	Naive Baseline	0.00	0.01	0.01	0.03	0.03	0.09
		Typilus	0.32	0.43	0.41	0.53	0.43	0.55
		HiTyper	**0.43**	**0.55**	**0.52**	**0.63**	**0.56**	**0.67**

rare types, HiTyper outperforms Type4Py and Typilus by 31% and 34%, respectively, w.r.t. the top-1 exact match metric. Regarding the prediction of user-defined types, HiTyper increases the performance of Type4Py and Typilus by 69% and 47%, respectively.

Answer to RQ2: HiTyper greatly alleviates the prediction issue of rare types faced by DL models by achieving a $> 30\%$ boost, taking advantage of the static type inference component.

3.4.4 RQ 3: Performance of the Static Type Inference Component

In this RQ, we evaluate the performance of the static type inference component in HiTyper compared with popular static type inference tools Pytype [166] and Pyre [160]. The results are shown in Table 3.5. We only consider the type categories of argument and return value for comparison since Pyre and Pytype do not infer types for local variables. We use the metric *number of correct anno-*

Table 3.5: Comparison with static type inference tools.

Dataset	Type Category	Approach	Exact Match	#Correct Annotations
ManyTypes4Py	**Argument**	Pytype	-	0
		Pyre Infer	**0.96**	613
		HiTyper	0.94	**1060**
	Return Value	Pytype	0.81	777
		Pyre Infer	0.84	662
		HiTyper	**0.86**	**2603**
	All	Pytype	0.81	777
		Pyre Infer	**0.89**	1275
		HiTyper	0.88	**3663 (16918*)**
Typilus's Dataset	**Argument**	Pytype	-	0
		Pyre Infer	**0.96**	543
		HiTyper	0.88	**983**
	Return Value	Pytype	0.79	552
		Pyre Infer	0.71	484
		HiTyper	**0.91**	**2461**
	All	Pytype	0.79	552
		Pyre Infer	0.82	1027
		HiTyper	**0.90**	**3444**

* The number of correct annotations when including local variables.

tations to replace the metric *match to parametric* that is usually used to evaluate DL models, considering that the results of static inference are exact and not recommendations.

As shown in Table 3.5, the exact match scores of all the static tools are greatly high, and HiTyper achieves the best performance. The results indicate the effectiveness of the static type inference component in HiTyper. We also find that there remains ∼10% of the results inconsistent with human annotations in

the datasets. By using Python's official type checker *mypy* to check these results, we observe that all the types annotated by HiTyper do not produce type errors, which reflects the correctness of the proposed HiTyper. After manual checking of these inconsistent types, we find this inconsistency is caused by subtypes, we further discuss them in Sec. 3.5. Besides, *mypy*'s results indicate very few inconsistent cases are caused by incorrect human annotations. To test whether HiTyper can rectify the incorrect annotations, we replace the original annotations with the results inferred by HiTyper, and inspect whether the original type errors are fixed. We finally correct 7 annotations on 6 GitHub repositories, including memsource-wrap [53], MatasanoCrypto [6], metadata-check [225], coach [79], cauldron [190], growser [201], and submit pull requests to these repository owners. The owners of MatasanoCrypto and cauldron have approved our corrections.

While Pytype and Pyre present high exact match scores, the numbers of variables they can accurately infer are small. Table 3.5 shows that HiTyper generally outputs 2x argument types and 3x return value types compared with them in both datasets, which suggests HiTyper's stronger inference ability than Pyre and Pytype. Such improvements attribute to HiTyper's import analysis and [Class Instantiation] rule on supporting the inference of user-defined types, and inter-procedural analysis on supporting the inference of class attributes and functions.

Answer to RQ3: Only considering the static inference part, HiTyper still outperforms current static type inference tools by inferring $2\times \sim 3\times$ more variables with higher accuracy.

3.5 Discussion

Inference of subtypes. Although HiTyper achieves promising results for type prediction and passes the check of *mypy*, it is still unable to infer some variable types (around 10%). The failure mainly occurs in the inference of subtypes.

```
1  #File: miyakogi.wdom/wdom/node.py
2  #Human annotation: AbstractNode
3  #Typilus: ForeachClauseNode      HiTyper: Node
4  def _append_element(self, node: AbstractNode) -> AbstractNode:
5      if node.parentNode:
6          node.parentNode.removeChild(node)
7      self.__children.append(node)
8      node.__parent = self
9      return node
10 def _append_child(self, node):
11     if not isinstance(node, Node):
12         raise TypeError
13     ...
14     return self._append_element(node)
```

Listing 3.2: An example HiTyper fails to infer.

Listing 3.2 shows an example for which HiTyper's result is inconsistent with the original annotations but still passes the check of *mypy*. The return statement at Line 9 indicates that the type of return value is the same as the type of argument `node`. Typilus predicts the type as `ForeachClauseNode`, which is invalid since it is not imported in the code and is from other projects in the training set. HiTyper infers the type as `xml.dom.Node`, because the function is called by another function named *_append_child* in the same file, and the caller transmits a variable with type `Node`. However, developers annotate the variable as `AbstractNode`, the parent type. Such behavior is common in practice and

poses a challenge for accurate type prediction.

3.6 Summary

In this chapter, we propose HiTyper, a hybrid type inference framework that iteratively integrates DL models and static analysis for type inference. HiTyper creates TDG for each function and validates predictions from DL models based on typing rules and type rejection rules. Experiments demonstrate the effectiveness of HiTyper in type inference, enhancement for predicting rare types, and advantage of the static type inference component in HiTyper.

Chapter 4

Generative Type Inference for Python

In the previous chapter, we introduce HiTyper, which combines static type inference and neural type predictions for more accurate and effective type inference. However, the performance upper bound of HiTyper is limited by the effectiveness of the deep learning model used in HiTyper. In this chapter, we focus on how to improve the performance of data-driven type inference approaches. The main points in this chapter are as follows. (1) We propose a few-shot generative type inference approach named TypeGen for Python. (2) We propose a novel prompt design to incorporate different static domain knowledge into language models, which includes code slices, type hints, and COT prompts. (3) We conduct extensive experiments to demonstrate the effectiveness of TypeGen compared with supervised and cloze-style type inference approaches, as well as the capability of TypeGen on language models with different parameter sizes.

4.1 Introduction

With the boom of artificial intelligence and data science, Python is becoming increasingly popular in recent years. As a dynamically typed programming language, Python is famous for its convenience and usability. The dynamic type system makes it possible to reuse the same code snippets for different functionalities, which significantly improves development efficiency. However, this convenience comes with a cost. The dynamic type system poses a threat to the reliability of Python software by introducing more type errors. Oh *et al.* [143] find that 30% of questions raised by developers at GitHub and Stack Overflow are related to type errors. To reduce potential type errors, Python Software Foundation introduces type annotations in a series of Python Enhancement Proposals (PEPs) [99, 100, 204, 250]. Manually annotating types for each variable in Python programs is overwhelming, so many automatic type inference approaches [7, 68, 128, 153, 157] are proposed to infer types statically to release the burden of developers. Automatic type inference approaches work along with static type checkers [133, 160, 161, 166] to detect potential type errors for Python programs [143, 155].

The earliest proposed automatic type inference approaches are rule-based, as described in previous work [10, 20, 29, 46, 80, 151]. These approaches rely on pre-defined typing rules and static analysis to accurately infer types. However, they are limited by the low coverage problem since the types of many variables in Python programs cannot be resolved statically.

Inspired by the remarkable achievements of deep learning in the natural language processing (NLP) field, supervised type inference approaches [7, 81, 128, 157, 220] take the code context of the target variable as input and leverage deep learning models to classify the context into one type. They can naturally avoid the low coverage problem of rule-based approaches, as deep learning models are

feature-agnostic and make predictions based on probability rather than rules. Taking advantage of identifiers in code, supervised type inference approaches are quite effective after training on a large dataset of annotated code. Despite the effectiveness, supervised type inference approaches based on classification methods classify inputs into pre-defined type categories and perform badly on rare types. Peng *et al.* [152] propose a hybrid type inference approach HiTyper to mitigate these problems by using deep learning models to recommend type predictions for static analysis. However, deep learning models in supervised approaches require large high-quality datasets of type annotations, which needs substantial human efforts.

Cloze-style type inference approaches [31, 44, 61, 62, 217] transform the type inference problem into a fill-in-the-blank problem by adding masks on the locations of type annotations in code. These approaches are well-aligned with the pre-training objectives of pre-trained code models and do not require large datasets, making them suitable for zero-shot settings. However, they face the following challenges:

1) Lack of static domain knowledge. Cloze-style approaches are characterized by the insertion of masks in source code, allowing pre-trained code models to predict the missing type information. While they have the advantage of not requiring large datasets, unlike supervised approaches, they rely solely on the general knowledge that pre-trained code models acquire during the pre-training phase. Consequently, their performance may be suboptimal, as they lack an understanding of how types are constructed based on typing rules.

2) Lack of interpretability. Current learning-based type inference approaches, including supervised and cloze-style approaches, adopt the *input-output* methodology, taking code as input and outputting single types. However, they provide no idea about how deep learning models reach the *output* types from the *input*

code. This lack of transparency makes it challenging for developers to comprehend and validate the predicted types, particularly when there are insufficient static constraints.

Our work. We propose TypeGen, the first few-shot generative type inference approach for Python programs. TypeGen has four phases, including code slicing, type hint collection, chain-of-thought (COT) prompt construction, and type generation. In the code slicing phase, TypeGen generates type dependency graphs (TDGs) and builds code slices based on TDG as the contexts of target variables, i.e., the variables whose types need to be inferred. In the type hints collection phase, TypeGen collects all available user-defined types and third-party types as type hints via import analysis to provide additional knowledge that does not exist in code slices. In the COT prompt construction phase, TypeGen translates the inference steps of static analysis for target variables into a COT prompts [218]. The code slices, type hints and COT prompts generated in the first three phases are combined as the *example prompts*, which provides rich static domain knowledge. In the last type generation phase, TypeGen adopts the *in-context learning* (ICL) methodology and constructs the input prompt by concatenating several example prompts and the *target variable prompt*, which includes code slice as well as type hints of the target variable. A language model is then invoked to complete the input prompt with the COT prompt of the target variable. With both explanations and predicted types in the generated COT prompts, TypeGen can improve the interpretability of results.

We evaluate TypeGen on the widely-used ManyTypes4Py dataset [127]. Our experiment results show that TypeGen outperforms the most advanced baseline Type4Py [128] by 10.0% for argument types and 22.5% for return value types in terms of top-1 Exact Match. Furthermore, we observe that TypeGen can achieve improvements of 27% $\sim$ 84% over the zero-shot performance of lan-

guage models with parameter sizes ranging from 1.3B to 175B in terms of top-1 Exact Match, which are $2\times \sim 3\times$ of the improvements achieved by the standard ICL method without static domain knowledge.

Contributions. We summarize our contributions as follows.

- To the best of our knowledge, we propose the first few-shot generative type inference approach named TypeGen for Python.
- We propose a novel prompt design to incorporate different static domain knowledge into language models, which includes code slices, type hints, and COT prompts.
- Extensive experiments demonstrate the effectiveness of TypeGen compared with supervised and cloze-style type inference approaches, as well as the capability of TypeGen on language models with different parameter sizes.

4.2 TypeGen

4.2.1 Overview

As a generative approach, TypeGen first generates domain knowledge-aware prompts and then inputs them into language models for type prediction. To achieve this, TypeGen adopts the widely-used *in-context learning* methodology [26]. This methodology provides a few example questions and answers as demonstrations for the language model and then asks the answer for a new question. Leveraging this methodology, TypeGen constructs the input prompt by adding some domain knowledge-aware *example prompts* (example questions and answers) before the *target variable prompt* (new question). Fig. 4.1 illustrates an input prompt with an example from the code in Fig. 4.3. The domain-aware

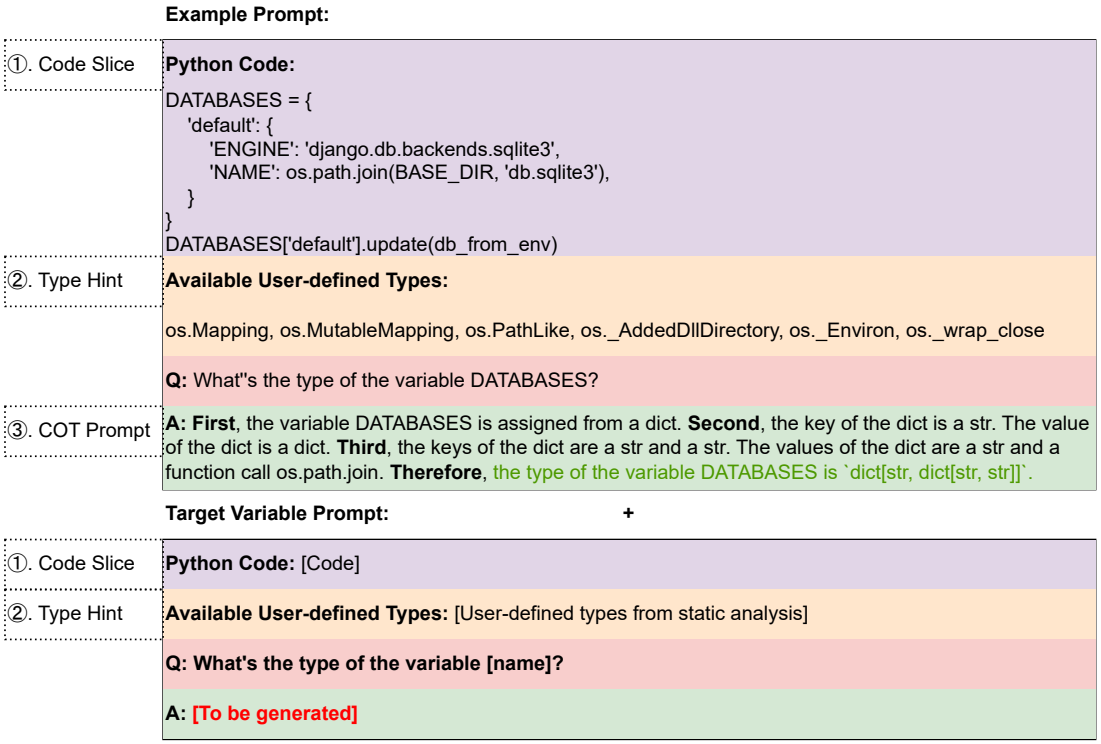


Figure 4.1: Input prompt with example from the code in Fig. 4.3.

example prompts include three parts: code slice, type hint and COT prompt, as shown in Fig. 4.1. They are designed to incorporate different static domain knowledge for language models. Specifically, the **code slice** isolates the statements contributing to the construction of the type for the target variable, with the remaining unrelated statements removed. The **type hint** includes external knowledge that is specific to different code slices, including user-defined types and third-party types. The **COT prompt** indicates the inference steps of static analysis, aiming at teaching language models how to infer types. The *target variable prompt* contains only the code slice and the type hint of the target variable.

We provide an overview of TypeGen's workflow in Fig. 4.2. To start, TypeGen takes a set of annotated Python source files to select examples and a target Python source file where the target variable is located. For each target source file,

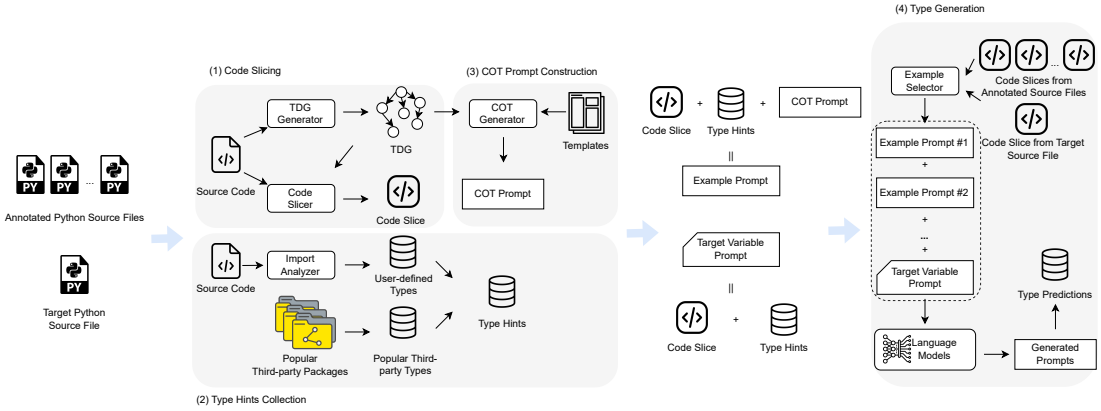


Figure 4.2: The overview of TypeGen.

TypeGen generates an input prompt that incorporates domain-aware example prompts and the target variable prompt. A language model is then employed to produce the COT prompt, which includes both the predicted type and corresponding explanations. To generate domain-aware example prompts, TypeGen conducts three phases: (1) *code slicing*, (2) *type hint collection*, and (3) *COT prompt construction* to generate the code slice, type hint, and COT prompt, respectively, finally, in the (4) *Type Generation* phase, TypeGen leverages in-context learning to infer the types of target variables.

4.2.2 Code Slicing

The code slicing phase aims at identifying the code statements related to the target variable based on the type dependency graph (TDG), which indicates the type dependencies among variables [153].

TDG Generation

In order to extract the type dependencies of the target variable, TypeGen generates type dependency graphs (TDGs) using HiTyper [153]. A TDG is a

```
...
12 import os
...
25 DEBUG = bool( os.environ.get('DJANGO_DEBUG', True) )
27 ALLOWED_HOSTS = ['stepper-v2.herokuapp.com', '127.0.0.1']
...
71 DATABASES = {
72    'default': {
73        'ENGINE': 'django.db.backends.sqlite3',
74        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
75    }
76 }
...
129 db_from_env = dj_database_url.config(conn_max_age=500)
130 DATABASES['default'].update(db_from_env)
...
```

Figure 4.3: The source code for the example prompt in Fig. 4.1, where DATABASES is the target variable.

directed graph (N, E), where N is the node set and E is the edge set. Each node $n \in N$ in TDG represents a variable (symbol node), an operation (operation node), or a type (type node), while each edge $e \in E$ indicates that the type of the output node depends on the type of the input node. If the target variable is an argument, a return value or a local variable in the function, TypeGen generates the TDG for the specific function. Otherwise, if the target variable is a global variable, TypeGen generates the TDG for all statements in the source file except class definitions and function definitions. To better illustrate the code slicing phase, we give a code example in Fig. 4.3 and its sliced TDG in Fig. 4.4, where the target variable is "DATABASES".

To refine the initial TDG, TypeGen prunes the nodes that do not have any type dependency with the target variable. For instance, in the code presented in Fig.4.3, TypeGen removes the nodes generated from statements at lines 25, 27, 129, etc. TypeGen then merges identical symbol nodes that are

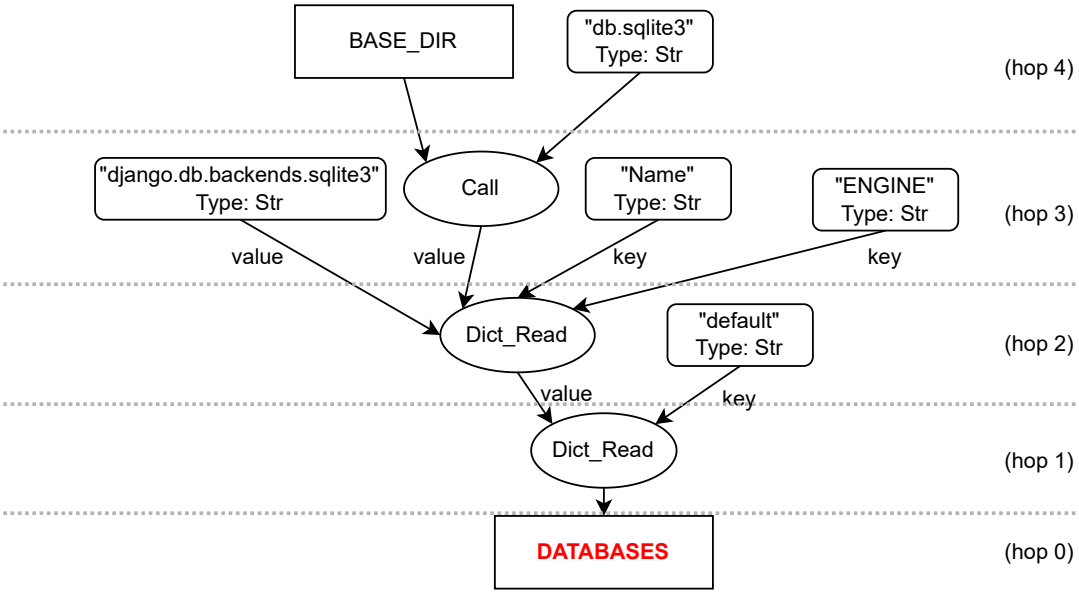


Figure 4.4: The sliced type dependency graph (TDG) of code in Fig. 4.3.

directly connected, since they represent different occurrences of the same variable. To generate the sliced TDG, TypeGen locates the sub-graph where the target variable is defined in the refined TDG. For the example in Fig.4.3, TypeGen identifies the target variable node "DATABASES" at line 71 and extracts the reachable sub-graph of the refined TDG as the sliced TDG, as illustrated in Fig. 4.4.

Code Slice Generation

Using the sliced TDG, TypeGen generates a code slice from the original input source code, containing only the statements that have type dependencies with the target variable. In this way, TypeGen reduces the entire function into a smaller code slice that includes only the information relevant to the type inference of the target variable. TypeGen employs different strategies for generating code slices for local variables, return values, and arguments.

Arguments. To facilitate the generation of code slices for function arguments, TypeGen adopts a different approach, since arguments do not have explicit definitions. TypeGen collects the usages of function arguments, as variable usages often provide hints about their types. For instance, operations such as *open* are usually associated with *File* types. Thus, TypeGen begins with all nodes of the target argument and traverses **forward** on the TDG, i.e., in the same direction as the TDG edges, to include nodes that use the target argument. TypeGen generates code slices for nodes with distances within a specified maximum threshold, similar to local variables and return values.

4.2.3 Type Hints Collection

Unlike built-in types that are available in every source file, user-defined and third-party types are specific to each source file and are defined through class definitions and import statements. However, general knowledge bases in language models do not cover this specific domain knowledge [153]. To address this issue, TypeGen collects all the user-defined and third-party types that are imported to the current source file as type hints.

To identify user-defined types, TypeGen performs an import analysis on the current source directory. First, it collects all class definitions in the current source file as user-defined types. Then, it examines the import statements to determine which source files are imported in the current file and adds their class definitions to the list of user-defined types. For third-party types, following previous study [241], TypeGen downloads the top 10,000 popular Python packages ranked by libraries.io [105] and employs the same import analysis technique to identify third-party types. All third-party types collected by TypeGen are stored in a database, which can be queried by TypeGen to identify the available third-party types based on the import statements in the current source file.

When generating type hints, TypeGen analyzes all the import statements in the current source file. If a user-defined package is imported, TypeGen directly conducts import analysis to gather all available user-defined types. If a third-party package is imported, TypeGen queries the database and obtains all available third-party types. All available types are concatenated to build the type hint, with an example shown in Fig. 4.1 (highlighted in orange color). To prevent excessively long type hints, TypeGen imposes a maximum threshold (set to 50 in this chapter) for the number of collected types. Considering the scarcity of user-defined types, TypeGen prioritizes the importance of types based on the following order: "user-defined types in current source file > user-defined types in other source files > third-party types.".

4.2.4 Chain-of-Thought Prompt Construction

Table 4.1: Chain-of-Thought Prompt Template. [NAME] indicates the name of symbol nodes, [OP] indicates the name of operation nodes and [TYPE] indicates the name of type nodes. [GTTYPE] indicates the annotated type for the target variable. DD-RV and DD-A indicate the dependency description for local variables and return values, and arguments, respectively.

Part	**Type**	**Template**
DD-RV	Operation→Symbol	The variable/return value of [NAME] is assigned from [OP] operation.
	Symbol→Symbol	The variable/return value of [NAME] is assigned from variable [NAME].
	Type→Symbol	The variable/return value of [NAME] is assigned from [TYPE].
	Operation→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are [OP] operation.
	Symbol→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are variable [NAME].
	Type→Operation	The operand(s)/target(s)/key(s)/value(s) of [OP] is/are [TYPE].
DD-A	Usage	The argument [NAME] is used in [OP]/[NAME].
	Naming	Based on the naming convention, it is reasonable to assume that the type of the argument [NAME] is [GTTYPE].
Con	Conclusion	Therefore, the type of the variable/return value of/argument [NAME] is [GTTYPE].

TypeGen translates the type inference steps of static analysis into chain-of-thought (COT) prompts [218] to involve static domain knowledge of how a type

is constructed, where a COT is a series of intermediate reasoning steps [218]. To generate COT prompts, TypeGen utilizes the sliced TDG produced by the *code slicing* phase.

Given the sliced TDG, TypeGen first organizes the nodes into different hops according to their distance from the target variable node. The hop of the node of the target variable is set to 0, and the hops of other nodes are determined by their distance, as shown in Fig. 4.4. The COT prompt constructed by TypeGen includes *dependency description* and *conclusion*. The *dependency description* explains how the type is inferred, whereas the *conclusion* gives the final type prediction. TypeGen translates each hop in the TDG into a sentence of *dependency description* and generates the *conclusion* based on the annotated types from developers. Following the recent study [245], we summarize the powerful templates of COT prompts from the zero-shot outputs of language models and present them in Table 4.1. For the *conclusion*, TypeGen fills in the variable name and annotated type into the template. Regarding the *dependency description*, TypeGen employs different prompt templates for local variables, return values, and arguments as TypeGen utilizes different information for them in the sliced TDGs in Sec. 4.2.2.

Local Variables and Return Values. Local variables and return values have clear definitions in the code, so it is possible to construct a comprehensive description of how the type of target variable should be inferred. To construct the *dependency description* for them, TypeGen starts with the edges connected to the target variable node and traverses backward on the TDG to translate each edge into a sentence of *dependency description*. Since there are three major types of nodes in the TDG, we design six templates for six kinds of edges in TDG to generate *dependency descriptions*, as shown in the first part of Table 4.1. Note that the type node cannot be the output node as its type does not depend on

other nodes. Take the TDG in Fig. 4.4 as an example. There is an edge from operation node *Dict_Read* to symbol node *DATABASES*. For this edge, TypeGen adopts the *Operation→Symbol* template and generates a sentence "*The variable DATABASES is assigned from a dict*". There is also an ordinal number at the beginning of each sentence to indicate one inference step. Sentences generated from all edges are concatenated together according to the backward traversal order to form a complete *dependency description*.

Arguments. As arguments usually do not have clear definitions in the code, it is difficult for static analysis to infer their types. Rather than providing solid definition information, TypeGen provides usage and naming information as hints for type prediction. We present the usage template and naming template in the second part of Table 4.1. For usage information, TypeGen collects all nodes in the sliced TDG and constructs the sentence "*The argument ... is used in ...*". For the naming information, TypeGen adds the sentence "*Based on the naming convention, it is reasonable to assume that the type of the argument is ...*" to remind language models to consider the argument name. These two sentences form a complete *dependency description* for arguments.

The generated *dependency description* and *conclusion* are finally combined together to form a complete COT prompt. Fig. 4.1 shows the COT prompt generated by TypeGen for the code example in Fig. 4.3, highlighted in green color.

4.2.5 Type Generation

For each input source file and target variable, TypeGen generates its corresponding code slice and selects a set of code slices from the training set as examples based on BM25 similarity [182]. BM25 similarity calculates the token similarity between two code slices and has been widely used in recent studies [50, 102].

Following previous work [107], the example prompts are ordered based on the BM25 similarity of code slices: $\{EP_1, ..., EP_n\}(\forall i \in [1, n)\ BM25(EP_i, TP) \leq BM25(EP_{i+1}, TP))$ where EP is an example prompt and TP is the target variable prompt. The example prompts are then combined with the target variable prompt to form the complete input prompt, as shown in Fig. 4.1.

Given the examples in the input prompt, language models can learn to generate a similar COT prompt for the target variable. To facilitate the automatic evaluation of the generated COT prompts, TypeGen surrounds type predictions in example COT prompts with quotes, such as ďict[str, dict[str, str]]ìn Fig. 4.1. This allows language models to learn to emphasize type predictions in generated COT prompts by adding quotes. Ultimately, TypeGen extracts the content within the quotes as the types predicted by the language models.

4.3 Experiment Setup

4.3.1 Dataset

We follow previous work [129, 153] and evaluate our approach on the ManyTypes4Py dataset [127] by splitting the dataset into a training set and a test set with an 80:20 ratio. We use the training set to train the baseline models and select examples for TypeGen and evaluate the performance of TypeGen and baselines in the test set. In order to accommodate the computational resource limitations, we further sample 10,000 instances from the test set during the evaluation of large language models. Table 4.2 presents the statistics of the experimental datasets.

Table 4.2: The statistics of the ManyTypes4Py dataset. 'Arg" indicates function arguments, "Ret" indicates function return values, "Var" indicates global and local variables, "Ele" indicates elementary types, "Gen" indicates generic types, and "Usr" indicates user-defined types and third-party types.

Dataset	Total	Arg	Ret	Var	Ele	Gen	Usr
Training	242,954	48,461	22,034	172,459	128,006	67,185	47,763
Set	100%	20.0%	9.1%	70.9%	52.7%	27.6%	19.7%
Test	85,205	16,700	7,754	60,751	44,605	23,310	17,290
Set	100%	19.6%	9.1%	71.3%	52.5%	27.4%	20.1%
Sampled	10,000	1,995	914	7,091	5,199	2,748	2,053
Test Set	100%	20.0%	9.1%	70.9%	52.0%	27.5%	20.5%

4.3.2 Baselines

We choose the following four type inference approaches as our baselines:

- **TypeBERT** [81] is a supervised type inference apporoach. It reformulates the type inference problem into a Named Entity Recognition (NER) problem and regards types as labels.

- **TyperWriter** [157] is a supervised type inference approach. It extracts different code features, such as identifiers and code tokens, and utilizes four RNNs to encode the extracted features and make type predictions.

- **Type4Py** [129] is a supervised type inference approach. It builds different type clusters and classifies a new Python program into one of the type clusters to determine the type.

- **HiTyper** [153] is a hybrid type inference approach. It builds a type dependency graph (TDG) for each target variable and utilizes both static analysis

Table 4.3: The statistics of language models used in the evaluation.

Model	#Parameters	Training Dataset	Type
CodeT5	220M/770M	CodeSearchNet & BigQuery	Generative & Infilling
UniXcoder	126M	CodeSearchNet	Infilling
GPT-Neo	1.3B/2.7B	The Pile	Generative
InCoder	1.3B/6.7B	GitHub & Stack Overflow	Generative & Infilling
CodeGen	6B	The Pile, BigQuery & GitHub	Generative
GPT-J	6.7B	The Pile	Generative
GPT-3.5	175B	-	Generative
ChatGPT	175B	-	Generative

and neural prediction to fill the blanks in the TDG and finally outputs the validated types.

We choose three popular pre-trained code models, including CodeT5 [217], UniXcoder [61], and InCoder [44] to represent the performance of cloze-style type inference approaches. Besides, we choose GPT-Neo [12], GPT-J [206], CodeGen [140], GPT-3.5 [14] and ChatGPT [145] to evaluate the performance of TypeGen on language models with different parameter sizes. We present the statistics of all the language models in Table 4.3.

4.3.3 Metrics

We use two commonly used metrics in previous work [7, 129, 153] to evaluate the performance of TypeGen and other baselines:

- **Exact Match** is defined by the ratio of type predictions made by an ap-

proach that **exactly** match type annotations from developers.

- **Match to Parametric** is defined by the ratio of type predictions made by an approach that **share the common outmost type** with type annotations from developers.

For example, *List[int]* and *List[str]* are considered Match to Parametric but not Exact Match since they are different types while sharing the same outmost type *List*.

4.3.4 Implementation

For the four type inference baselines TypeBERT [81], TyperWriter [157], Type4Py [129] and HiTyper [153], we directly use the replication packages released by the authors and other researchers. For all the language models except GPT-3.5 and ChatGPT, we download them from HuggingFace Hub [77] and deploy them locally. Following the previous work [129, 153], we adopt the generated sentences with top-5 probabilities as predictions. For GPT-3.5 and ChatGPT, we use the public APIs provided by OpenAI under engine "*text-davinci-003*" and "*gpt-3.5-turbo-0301*", respectively. We acquire 50 samples with a temperature of 1.0 for each target variable and rank the top-5 predictions according to the occurrence frequency, following the work [226, 229]. We choose the maximum distance threshold of TDG nodes at 3 for the code slicing phase of TypeGen, as previous studies [7, 129] only consider types with nested levels smaller than 3. All experiments are conducted on a Linux machine (Ubuntu 18.04) with one 112-core Intel Xeon Gold 6348 CPU@ 2.60GHz, two NVIDIA A100-80GB GPUs, and 1TB RAM.

4.4 Evaluation

4.4.1 Research Questions

In the evaluation, we focus on the following four research questions:

- **RQ1**: How effective is TypeGen in type inference compared with existing approaches?
- **RQ2**: How capable is TypeGen in language models with different parameter sizes?
- **RQ3**: What are the impacts of different parts in the prompt design of TypeGen?
- **RQ4**: What are the impacts of different examples in TypeGen?

To study RQ1, we conduct experiments on both TypeGen and baseline approaches with the entire test set (85,205 instances), aiming to comprehensively verify the effectiveness of TypeGen against state-of-the-art type inference techniques. However, due to limited computational resources, we only use the sampled test set (10,000 instances) for RQ2-4. For RQ2, we evaluate six language models under TypeGen to examine the tool's effectiveness across language models with different parameter sizes. We also include two additional settings: **Zero-Shot** and **Standard ICL**. In the **Zero-Shot** setting, we do not provide any example and only use the source code of the target variable as the input prompt for language models in the type prediction. The Zero-Shot setting tests the basic performance of language models on type prediction. In the **Standard ICL** setting, we provide three fixed example prompts before the target variable prompt and use only source code in the input prompts, which is the same with recent study [103]. The Standard ICL setting indicates the basic performance of language models

with the *in-context learning* methodology. To fairly compare TypeGen with the Standard ICL setting, we also set the number of examples in TypeGen to 3 in RQ2. For RQ3, we remove different parts of the prompt design in TypeGen to study the impacts of each part. For RQ4, we vary the number of examples and the example selection method to investigate the impacts of different examples. We choose ChatGPT as the base model of TypeGen and use five examples in TypeGen in all the RQs except RQ2.

Table 4.4: The performance of TypeGen along with the baselines under four types of variables in terms of Top-1,3,5 Exact Match (%) and Match to Parametric (%). "Arg", "Ret", "Var", and "All" indicate function arguments, function return values, global and local variables, and all of above, respectively. Under each metric the best performance is marked as gray.

Metric	Category	Approach	Top-1				Top-3				Top-5			
			Arg	Ret	Var	All	Arg	Ret	Var	All	Arg	Ret	Var	All
Exact Match (%)	Supervised	TypeBERT	28.0	38.5	51.1	45.4	34.8	52.6	55.8	51.4	36.5	57.1	58.6	54.1
		TyperWriter	53.3	52.8	-	-	61.1	60.7	-	-	65.8	65.3	-	-
		Type4Py	66.5	56.1	82.0	76.6	72.0	59.2	83.8	79.3	73.8	60.7	84.3	80.1
	Cloze Style	InCoder-1.3B	20.9	20.5	15.1	16.7	21.3	20.8	15.5	17.1	21.3	21.0	15.6	17.2
		InCoder-6.7B	24.1	42.0	18.7	21.9	24.6	42.7	19.1	22.3	24.7	43.1	19.2	22.4
		UniXcoder	55.0	49.2	35.9	40.9	66.9	64.6	42.1	49.0	70.6	69.8	45.2	52.4
		CodeT5-base	51.1	57.6	21.7	30.7	59.3	64.4	28.0	37.4	62.0	66.9	30.7	40.1
		CodeT5-large	56.2	60.2	44.7	48.4	61.6	64.5	50.4	53.9	63.9	66.3	53.4	56.6
	Generative	TypeGen	73.1	68.7	82.2	79.2	81.0	77.1	87.9	85.6	82.7	79.1	89.1	87.0
Match to Parametric (%)	Supervised	TypeBERT	29.8	41.4	54.0	48.1	36.0	55.9	58.0	53.5	37.7	60.8	61.2	56.5
		TyperWriter	54.4	54.1	-	-	63.4	63.5	-	-	68.8	69.3	-	-
		Type4Py	68.0	59.0	86.2	80.2	74.1	64.1	88.3	83.3	75.9	66.3	88.8	84.3
	Cloze Style	InCoder-1.3B	22.9	22.8	18.7	19.9	23.3	23.1	19.1	20.3	23.4	23.3	19.2	20.4
		InCoder-6.7B	28.8	51.6	25.0	28.1	29.3	52.1	25.3	28.5	29.4	52.5	25.3	28.6
		UniXcoder	61.9	61.8	44.3	49.3	72.3	76.0	51.2	57.6	75.0	80.1	53.8	60.4
		CodeT5-base	54.8	66.7	27.7	36.6	62.9	74.2	34.4	43.6	65.6	76.4	37.1	46.3
		CodeT5-large	61.4	69.4	55.7	58.0	66.8	74.3	61.2	63.5	68.9	76.2	63.7	65.9
	Generative	TypeGen	78.7	75.6	91.2	87.3	84.9	83.0	93.7	91.0	86.1	84.5	94.1	91.7

4.4.2 RQ1: Effectiveness of TypeGen

Comparison with Supervised Approaches

We compare TypeGen with three supervised type inference approaches, namely TypeBERT, TyperWriter, and Type4Py. The results are presented in Table 4.4, where we report the top-1, top-3, and top-5 Exact Match and Match to Parametric for four categories of variables. It is worth noting that TyperWriter is designed solely for argument and return value type predictions. Analyzing the top-1 prediction results in Table 4.4, we observe that TypeGen outperforms the best supervised approach, Type4Py, by 10.0% for argument type prediction and 22.5% for return value type prediction in terms of Exact Match. This improvement of TypeGen over Type4Py is even more significant for top-5 predictions, where TypeGen outperforms Type4Py by 12.1% for argument type prediction and 30.3% for return type prediction in terms of Exact Match. Moreover, when considering Match to Parametric, TypeGen achieves a consistent improvement of 8.7% $\sim$ 9.3% on overall variables than Type4Py. These results demonstrate that, even with few annotated examples, the generative type inference approach TypeGen is more effective than supervised approaches such as Type4Py. We also observe that TypeGen does not perform much better than Type4Py on predicting local variables. This can be attributed to the lower difficulty of type inference for local variables compared to arguments and return values, so static domain knowledge incorporated by TypeGen provides limited improvements. This can also be verified by Table 4.4, where all the supervised approaches obtained much higher performance on local variables than arguments and return values.

Table 4.5: The performance of HiTyper with different base models under four types of variables. Variable categories are the same with Table 4.4.

Metric	Base Model	Arg	Ret	Var	All
Exact Match (%)	-	8.0	43.5	65.7	52.4
	Type4Py	73.5	73.4	90.6	85.7
	TypeGen	84.9	77.9	90.5	88.3
Match to Parametric (%)	-	8.4	52.7	70.2	56.5
	Type4Py	76.1	83.4	95.3	90.1
	TypeGen	87.4	87.3	95.3	93.1

Comparison with Cloze-Style Approaches

We compare TypeGen with cloze-style approaches, namely InCoder, UniXcoder and CodeT5, and present the results in Table 4.4. Our observations indicate that, in general, cloze-style approaches perform worse than supervised approaches due to their lack of domain knowledge from data and static analysis. By introducing five annotated examples and incorporating static knowledge, TypeGen outperforms the best cloze-style approach CodeT5-large by 63.6% on overall top-1 Exact Match and 53.7% on overall top-5 Exact Match. This suggests that incorporating domain knowledge from static analysis with a few examples can largely improve the performance of type inference.

Comparison in Hybrid Approach HiTyper

As HiTyper is a hybrid approach, we study its performance with the best supervised approach Type4Py and TypeGen, and present the experiment results in Table 4.5. For Type4Py and TypeGen, we use their top-5 predictions as type recommendations since HiTyper can reject wrong types. The results show that

HiTyper performs poorly when there is no base model, particularly in argument type inference, where it achieves an Exact Match of only 8%. This verifies the low coverage problem of static analysis. When associating with base models, HiTyper with TypeGen still outperforms HiTyper with Type4Py by 15.5% for argument type inference and 6.1% for return value inference. This indicates that the performance gap of TypeGen over Type4Py cannot be bridged by simply combining them with static analysis.

Answer to RQ1: TypeGen outperforms the best baseline Type4Py by 8.6% on all variables, with particularly notable improvements of 12.1% and 30.3% for argument and return value types, respectively, in terms of top-5 Exact Match.

4.4.3 RQ2: Capability of TypeGen in Different Language Models

We compare the performance of TypeGen on six language models with parameter sizes ranging from 1.3B to 175B with the Zero-Shot setting and the Standard ICL setting and present the overall top-1,3,5 Exact Match in Table 4.6. In the Zero-Shot setting, our results indicate that language models with larger model sizes generally perform better, with ChatGPT achieving a 2× top-1 Exact Match than GPT-Neo-1.3B. When providing language models with three fixed examples in the Standard ICL setting, we observe an 8% $\sim$ 56% improvement in top-1 Exact Match, demonstrating the effectiveness of the *in-context learning* methodology. For TypeGen, we find consistent improvements of 27% $\sim$ 84% on different language models, with the improvements being more significant for smaller language models like GPT-Neo-1.3B than larger language models like ChatGPT. With less general knowledge stored in the models, smaller language models benefit more from the domain knowledge associated by TypeGen. Fur-

Table 4.6: The performance of different language models under three settings for all variables in terms of Top-1,3,5 Exact Match (%). △ indicates the improvement of Standard ICL and TypeGen over the Zero-Shot setting.

Base Model	Approach	Top-1 (△)	Top-3 (△)	Top-5 (△)
GPT-Neo (1.3B)	Zero-Shot	31.5	40.6	42.8
	Standard ICL	44.0 (40%)	50.0 (23%)	50.8 (19%)
	TypeGen	57.0 (81%)	61.5 (51%)	62.8 (47%)
GPT-Neo (2.7B)	Zero-Shot	43.2	50.0	51.9
	Standard ICL	46.6 (8%)	52.3 (5%)	52.8 (2%)
	TypeGen	55.5 (28%)	61.9 (24%)	63.0 (21%)
GPT-J (6.7B)	Zero-Shot	42.4	43.7	43.9
	Standard ICL	50.8 (20%)	54.9 (26%)	55.3 (26%)
	TypeGen	62.7 (48%)	67.3 (54%)	68.4 (56%)
CodeGen (6B)	Zero-Shot	34.7	44.0	45.5
	Standard ICL	54.1 (56%)	60.5 (38%)	61.9 (36%)
	TypeGen	63.7 (84%)	69.1 (57%)	70.8 (56%)
GPT-3.5 (175B)	Zero-Shot	62.0	65.4	66.3
	Standard ICL	69.7 (12%)	74.2 (13%)	75.8 (14%)
	TypeGen	78.9 (27%)	85.0 (30%)	86.2 (30%)
ChatGPT (175B)	Zero-Shot	61.3	66.1	67.5
	Standard ICL	68.0 (11%)	71.8 (9%)	73.1 (8%)
	TypeGen	78.8 (29%)	85.3 (29%)	86.7 (28%)

thermore, the improvements achieved by TypeGen over the Zero-Shot setting are $2\times \sim 3\times$ of that achieved by the Standard ICL setting, in terms of top-1 Exact Match. For the top-5 type prediction, TypeGen even achieves a $10\times$ of improvement obtained by the Standard ICL setting on GPT-Neo-2.7B. These findings demonstrate the usefulness of incorporating static domain knowledge in the prompt design of TypeGen, which cannot be outweighed by simply providing some examples.

Table 4.7: The performance of TypeGen when removing different parts of the prompt design in TypeGen in terms of Top-5 Exact Match (%). "Ele", "Gen", and "Usr" indicate elementary types, generic types and user-defined types as well as third-party types, respectively. Other variable categories are the same with Table 4.4.

Ablation	**Arg**	**Ret**	**Var**	**Ele**	**Gen**	**Usr**	**All**
w/o Code Slice	74.8	77.0	68.8	75.1	75.5	73.9	70.8
w/o Type Hint	76.1	75.9	89.3	94.1	77.2	75.9	85.5
w/o COT Prompt	82.3	78.6	86.4	92.9	70.8	84.3	84.9
TypeGen	83.5	79.4	89.7	94.3	77.8	84.6	87.5

Answer to RQ2: TypeGen is capable of consistently improving the zero-shot performance of type inference for language models with different parameter sizes and achieves $2\times \sim 3\times$ of improvements made by the Standard ICL setting.

4.4.4 RQ3: Impacts of Different Parts of Prompt Design

To investigate the impact of different parts of the prompt design of TypeGen, we conduct an ablation study and present the results in Table 4.7. The results show that removing code slicing techniques and inputting the whole function of target variables in the prompts leads to a significant performance drop of 24% on overall type inference. This decrease is mainly caused by local variables and arguments, as there is typically only a small set of statements in the function that have type dependencies with them, while inputting the entire function can introduce useless information and bias the language model. When type hints are removed, the performance of TypeGen on user-defined types decreases the most

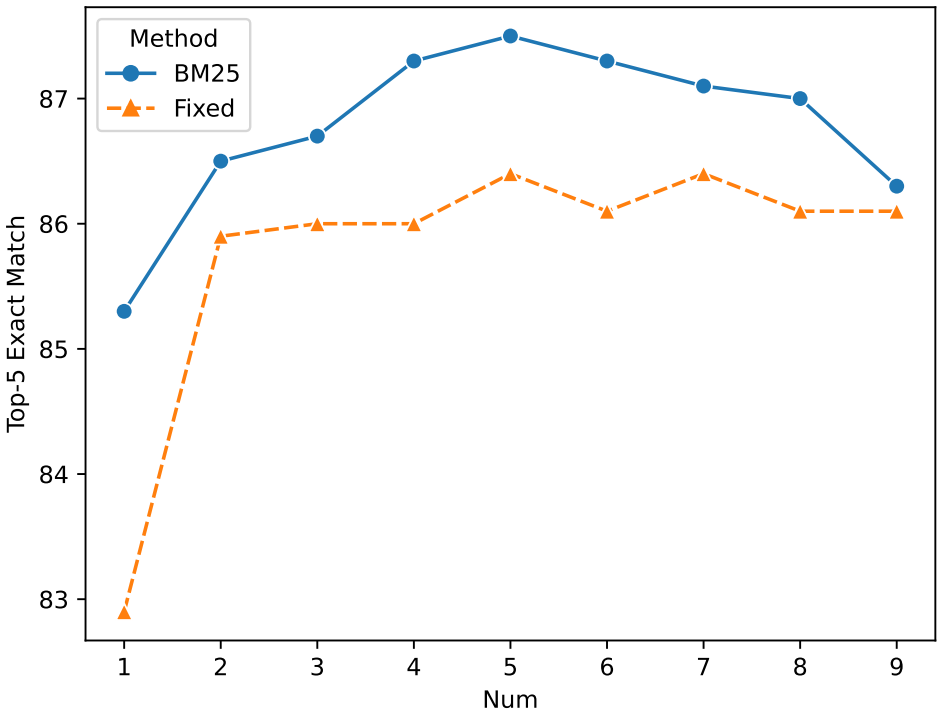


Figure 4.5: The top-5 Exact Match of TypeGen with different numbers of examples and different example selection methods

(11%), indicating the importance of providing available user-defined types as additional knowledge for language models. Additionally, when COT prompts are removed, the performance of TypeGen on generic types drops the most (10%), as generic types usually involve complicated type dependencies that should be well handled by static analysis. Providing the inference steps of static analysis in COT prompts can greatly help improve the performance of language models on generic types.

Answer to RQ3: In the prompt design, code slicing improves the overall performance of type inference by 24%, type hints improve the performance of user-defined type inference by 11%, and COT prompts improve the performance of generic type inference by 10%.

4.4.5 RQ4: Impacts of Different Examples

To evaluate the effects of the number of examples and example selection methods in the prompt design of TypeGen, we vary the number of examples from one to nine and compare two example selection methods: fixed examples and BM25 similarity-based examples. We present the top-5 Exact Match results of TypeGen in Fig. 4.5.

Based on the results presented in Fig. 4.5, we observe that the performance of TypeGen is largely affected by the number of examples provided in the input prompts. Specifically, the performance drops notably when there is only one example, highlighting the importance of providing sufficient examples for effective *in-context learning*. Additionally, the performance of TypeGen using BM25 similarity-based examples increases up to five examples, after which it starts decreasing. This suggests that both inadequate and excessively long input contexts can harm the performance of TypeGen. When changing the example selection method, we find that using BM25 similarity-based examples performs better than using fixed examples, particularly when only one example is provided. However, when we provide five examples, the performance drop with fixed examples is relatively small (less than 1.3%). One possible explanation is that large language models learn how to perform type inference from the example prompts rather than solely relying on the direct correlations between type predictions in different example prompts [126].

Python Function:

```
def compose_options():
    options = ["-f", compose_path("demo.yml")]
    return {"options": options, "name": "demo",
        "priority": ">base", "variant": "openedx"}
...
```

Generated COT Prompt:
First, the return value of compose_options is assigned from a dict. **Second**, the keys of the dict are a str, a str, a str, and a str. The values of the dict are options, a str, a str, and a str. **Third**, options is assigned from a list. **Therefore**, the type of the return value of compose_options is `dict[str, typing.Union[str,list[str]]]`.

Figure 4.6: A function whose return value type can only be inferred by TypeGen. The type annotation for the return value is Dict[str, Union[str, List[str]]].

Answer to RQ4: TypeGen achieves the best performance with five examples. TypeGen shows only a small performance drop (<1.3%) even when provided fixed examples, releasing the burden of developers to design examples.

4.5 Discussion

4.5.1 Interpretability of TypeGen

To better illustrate the interpretability of COT prompts generated by TypeGen, we give an example of a function whose return value can only be inferred by TypeGen in Fig. 4.6. Due to the page limitation, we only present the code slice and the generated COT prompt. To infer the return type of function *compose_options*, TypeGen follows similar inference steps as static analysis. First, it infers that the return value is assigned from a dictionary in the generated COT prompt. Then it identifies the types of keys and values by specifying that there are four keys in the dictionary with types of *str*, and there are three values with types of *str* as well as one variable named *options*. The second step in the gener-

ated COT precisely matches the code given in the input prompt, indicating that large language models like ChatGPT have the capacity to simulate the inference steps of static analysis. In the third step of the generated COT prompt, TypeGen recognizes the unknown variable *options* and locates its assignment from a list. Since we set the maximum number of hops to 3, TypeGen generates the conclusion directly after the third step. From this example, we can find that by providing an explanation in the COT prompt, human developers can easily understand the predictions and determine whether the predictions are correct based on the explanations.

4.5.2 Limitations of TypeGen

Despite the effectiveness of TypeGen, we also identify the following limitations:

1) *Limited context.* Although TypeGen adopts static code slicing techniques based on TDGs, we have observed a limited number of instances in the test set ($\sim$ 1000) with code slices exceeding the maximum context length of language models. This primarily occurs in extremely long functions with complex type dependencies. We recognize that extracting code slices without sacrificing key dependency information is still a challenge.

2) *Limited knowledge for function arguments.* As function arguments lack precise definitions, TypeGen provides naming and usage information to enable language models to predict their types. However, this information is incomplete and can potentially introduce biases in the model's predictions [49]. Incorporating data flow information via inter-procedural analysis may be a possible solution to enhance argument information.

4.6 Summary

In this chapter, we present TypeGen, a few-shot generative type inference method for Python programs. Our approach incorporates static domain knowledge into language models via a novel prompt design in the *in-context* learning paradigm. Experimental results show that TypeGen outperforms both state-of-the-art supervised type inference methods and cloze-style type inference methods.

Chapter 5

Domain-aware Prompt-based Automatic Repair for Python Type Errors

Type inference approaches can statically add type information to Python code, and thus prevent the occurrence of type errors by checking potential type errors. However, it provides limited help in fixing existing type errors. In this chapter, we focus on repairing existing type errors in the wild. The main points in this chapter are as follows. (1) We propose a domain-aware prompt-based approach for repairing Python type errors named TypeFix. (2) We propose a novel fix template design that can handle type errors at different levels, along with a novel hierarchical clustering approach to mine various fix templates from existing type error fixes. (3) We conduct extensive experiments to demonstrate the effectiveness of TypeFix compared with state-of-the-art rule-based and learning-based baselines, and the high coverage of the mined fix templates.

5.1 Introduction

Being used in most artificial intelligence and data science applications, Python has become extremely popular in recent years. According to GitHub Octoverse [56], which records the state of open-source software, Python is the second most-used programming language in 2022. Moreover, Python continues to see gains in its usage across GitHub with a 22.5% year-over-year increase [56].

Python adopts a dynamic type system, in which the type of a variable will be resolved only at run-time. This enables fast prototyping and brings much convenience for developers to write an executable program. The catch, however, is that more type errors occur at run-time, threatening the reliability of Python applications. Oh *et al.* [143] find that about 30% of questions in Stack Overflow and issues in GitHub of Python are about type errors. To avoid type errors, Python Software Foundation accepts several Python Enhancement Proposals (PEPs) [99, 100, 204, 250] and releases a static type checker named *mypy* [133], allowing developers to add type annotations and check potential type conflicts statically. What's more, the recent research [8, 130, 154, 195] on type inference aims at statically inferring the types of variables, which further reduces the burden of manual type annotation. These approaches can reduce potential type errors but provide limited help to repair existing type errors.

To automatically fix type errors, Oh *et al.* [143] propose the first rule-based approach. They manually define nine templates and several synthesis rules to generate patches via dynamic analysis, but the manually defined templates suffer from low coverage of real-world type errors, and designing patch synthesis rules requires substantial effort from domain experts.

General learning-based automatic program repair (APR) approaches [27, 86, 112, 228, 240, 249] have become quite popular and powerful in recent years since they are feature-agnostic and can automatically learn to generate patches

from existing bug fixes, without explicit definitions of synthesis rules. Among the learning-based approaches, the Neural Machine Translation (NMT)-based approach that translates the buggy lines into correct lines was typically used in the past. Most recently, Xia *et al.* [228] propose the first prompt-based APR approach named *AlphaRepair* and obtain state-of-the-art performance. Unlike NMT-based APR approaches, AlphaRepair transforms the APR problem into a fill-in-the-blank problem by masking several tokens in buggy lines and invoking pre-trained models to predict the masked tokens. Despite the superior performance of the prompt-based approach over NMT-based approaches, the prompts in AlphaRepair are pre-defined, i.e., where to mask and how to add masks in buggy code are manually designed. Without domain-specific knowledge, the prompt-based approach can hardly fix the type errors with complex patterns [143]. However, automatically incorporating the prompt with domain knowledge is challenging due to different levels of type errors and various type error fixing patterns.

Our Work. To address the aforementioned challenge, we propose TypeFix, a domain-aware prompt-based approach for repairing type errors. TypeFix has two main phases: the template mining phase and the patch generation phase. The template mining phase aims to extract and organize fix templates from existing type error fixes. Fix templates are designed to handle type errors at different levels (e.g., expression level and statement level). TypeFix first parses type error fixes into *specific fix templates* and then employs a novel hierarchical clustering algorithm to abstract and merge the *specific fix templates* into *general fix templates*. The patch generation phase aims at exploiting the mined fix templates in the first phase for producing patches. TypeFix selects the most matched and commonly-used fix templates based on Breadth-First Search (BFS) and a frequency-aware ranking algorithm, and then generates code prompts by apply-

ing the ranked fix templates, and invokes CodeT5 [217] for prediction. TypeFix is fully automated and extendable, as it does not need manually defined templates as well as patch synthesis rules. Additionally, the minded fix templates enable the proposed prompt-based TypeFix to be aware of domain knowledge when generating patches.

We evaluate TypeFix on two benchmarks BugsInPy [224] and TypeBugs [143] by comparing it with four baselines, including both the recent rule-based and learning-based approaches. In the BugsInPy benchmark, TypeFix successfully fixes 26 out of 54 type errors, outperforming the most effective baseline Codex [19] by 9. In the TypeBugs benchmark, TypeFix successfully fixes 55 out of 109 bugs, outperforming the most effective baseline PyTER [143] by 14. Experiments also show that the fix templates mined by TypeFix can cover about 75% of type errors in both benchmarks, much higher than PyTER which only covers 40% of the type errors. The results demonstrate the effectiveness of TypeFix in repairing Python type errors.

Contributions. We conclude our contributions as follows.

- To the best of our knowledge, TypeFix is the first domain-aware prompt-based approach for repairing Python type errors.
- We propose a novel fix template design that can handle type errors at different levels, along with a novel hierarchical clustering approach to mine various fix templates from existing type error fixes.
- Extensive experiments demonstrate the effectiveness of TypeFix compared with state-of-the-art rule-based and learning-based baselines and the high coverage of the mined fix templates.

5.2 Motivation

To better illustrate our motivation, we give an example in Listing 5.1. The type error in Listing 5.1 is from a popular GitHub project *scrapy* in the BugsInPy benchmark. The correct fix for this type error is to add a user-defined type conversion function *to_bytes* to the entire string, as shown in the green-colored line. We also provide the patches provided by the baseline approaches and TypeFix in Listing 5.1.

Baseline Approaches. CoCoNuT [112] is an NMT-based APR approach that translates the buggy line into the correct line. In the patch, it modifies the content of the string since the variables *user* and *password* are often used in authorization. However, this cannot fix the type error. AlphaRepair [228] is a prompt-based APR approach that masks the tokens in the buggy line to generate patches. In the patch, it masks the function name *unquote* and fills a new name *ascii* to generate the patch. Without the domain knowledge indicating there should be a new function call wrapping the entire buggy string, it fails to identify the correct location to add masks and thus fails to fix this type error. Codex [19] is a large language model from OpenAI. Powered by the huge knowledge base stored in the model, Codex identifies that this type error is related to *bytes* types, but it adds checks for the *user* and *password* instead of the entire string, failing to fix this type error. This may be because adding type conversions for variables is much more frequent than that for the entire expression. PyTER [143] is a rule-based approach via dynamic analysis. It fails to find the correct variable inducing the type error, and also cannot introduce the required user-defined type conversion function *to_bytes*.

```
# Buggy Code: scrapy/scrapy:f042ad
if user:
-     user_pass = '%s:%s' % (unquote(user), unquote(password))
```

```
+     user_pass = to_bytes('%s:%s' % (unquote(user), unquote(password)))
        raise ValueError('Port cannot be 0 or less.')
        creds = base64.b64encode(user_pass).strip()
else:
        creds = None
# Patches:
# Incorrect Patch provided by CoCoNuT
user_pass = 'Proxy-Authorization'%(unquote(user), unquote(password))
# Incorrect Patch provided by AlphaRepair
user_pass = '%s:%s' % (ascii(user), unquote(password))
# Incorrect Patch provided by Codex
if not isinstance(user, bytes):
    user = user.encode('ascii')
if not isinstance(password, bytes):
    password = password.encode('ascii')
# Incorrect Patch provided by PyTER
if isinstance(creds, bytes):
    creds = str(creds, 'utf-8')
# Correct Patch provided by TypeFix
user_pass = to_bytes('%s:%s' % (unquote(user), unquote(password)))
```

Listing 5.1: A type error in BugsInPy benchmark

TypeFix. Before fixing a type error, TypeFix first mines fix templates from existing type error fixes via hierarchical clustering. In the clustering process, TypeFix can generalize the fix pattern of adding a type conversion for a variable into that of adding a type conversion for an expression. Even though the later fix pattern has low occurrence frequency, TypeFix can still successfully identify and apply the fix pattern to this type error. Guided by the selected fix template, TypeFix adds a new function to wrap the original buggy string, and inserts masks for the name of the new function, instead of randomly masking several tokens. As a prompt-based APR approach, TypeFix also mitigates the problem

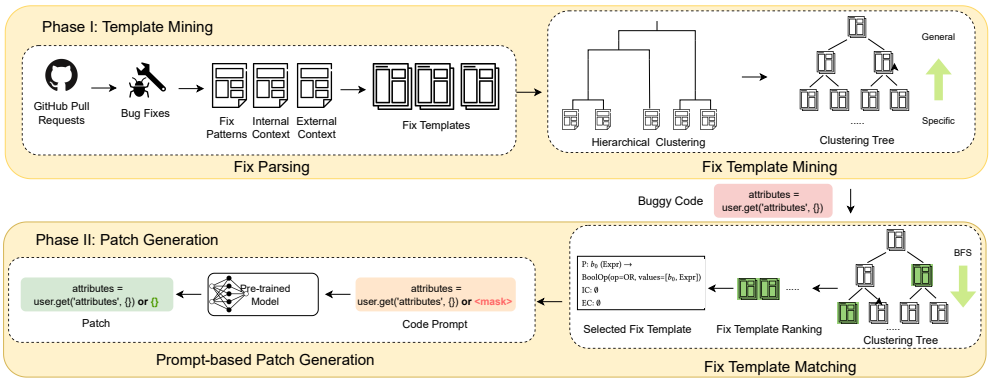


Figure 5.1: Overview of TypeFix

of introducing user-defined type conversion functions in rule-based approaches like PyTER, since language models can learn from the contexts of the type error. Therefore, TypeFix can successfully fix the type error.

5.3 TypeFix

5.3.1 Overview

TypeFix contains two main phases: the template mining phase and patch generation phase, with the overview shown in Fig. 5.1. In the *template mining* phase, TypeFix aims at extracting domain-aware fix templates. TypeFix first parses existing type error fixes into specific fix templates and then employs a novel hierarchical clustering algorithm to abstract and merge them into general fix templates. TypeFix also organizes the specific to general fix templates into *clustering trees*. In the *patch generation* phase, TypeFix aims at generating patches for new buggy programs by incorporating prompts with the mined fix templates. Specifically, it selects and ranks the mined fix templates and applies them to buggy code to generate domain-aware code prompts automatically. The CodeT5 [217] model is finally invoked to generate patches by filling the masks in

the code prompts.

5.3.2 Mining Phase

The template mining phase mainly contains two stages: fix parsing and fix template mining. The fix parsing stage aims to transform type error fixes into specific fix templates, and the fix template mining stage abstracts and merges parsed specific fix templates into general fix templates via the proposed hierarchical clustering algorithm. We first give formal definitions of fix templates for ease of understanding.

Definition of Fix Template

To represent the domain knowledge of where and how to add masks in buggy code for building code prompts, we define fix template as a combination of three parts: *fix pattern*, *internal context* and *external context*. The fix pattern indicates how the buggy code is edited to fix the type error, the internal context pinpoints the locations for applying fix patterns to handle type errors at different levels, and the external context indicates the location of the internal context in the entire buggy program. The three components are all represented based on *template trees*, which are defined below.

Definition 5.3.1. (Template Tree) A template tree is a tree (N, E, rt) with nodes N, edges E and root node $rt \in N$. An edge is a triple (n, n', r) where node n is the parent of n' with relation r. A node is a quadruple (bt, t, v, i) where $bt \in$ {Variable, Op, Literal, Builtin, Type, Attribute, Expr, Stmt} is the base type of node, t is the AST node type, v is the value, and i is the id. bt, t, and v have a special value ABS to represent a hole.

We define template trees based on the abstract syntax tree (AST) [40] of Python. Keeping the original AST node type t, we add a base type bt by re-

classifying all original AST node types and attribute types into eight base types, and thus a base type can include multiple AST node types, for example, AST node types *BoolOp*, *BinOp*, and *UnaryOp* belong to the same base type *Expr*. We design base types to obtain a higher level of abstraction than that of ASTs. For instance, the above three AST node types can all be the conditions of *If* statements that serve as guards to prevent type errors. Representing the three AST node types as *Expr* to indicate general conditions help create more general fix templates.

Definition 5.3.2. (Fix Pattern) A fix pattern is a map $B\_Tree \rightarrow A\_Tree$, where B_Tree is a template tree of the buggy code, and A_Tree is a template tree of the fixed code.

Definition 5.3.3. (Internal Context) An internal context is a pair (IC_Tree, rn), in which IC_Tree is a template tree of the deepest statement where a fix pattern locates, and rn is a map $n \rightarrow (br, ar)$ where br and ar are edge relations, $n \in IC\_Tree.N$ indicates the node where B_Tree is removed with the edge $(n, B\_Tree.rt, br)$ and A_Tree is added with the edge $(n, A\_Tree.rt, ar)$.

The internal context is defined to handle edits at different levels. For example, some expression-level edits only modify single expressions in the statements, while other statement-level edits replace the entire statements. Since fix patterns only represent the edits themselves, we use internal contexts to represent the rest parts of the deepest statements for expression-level edits. The internal contexts are empty when the edits are at the statement level.

Definition 5.3.4. (External Context) An external context is a pair (BC_Tree, AC_Tree), where BC_Tree is a template tree of statements before the internal context and AC_Tree is a template tree of the statements after the internal context.

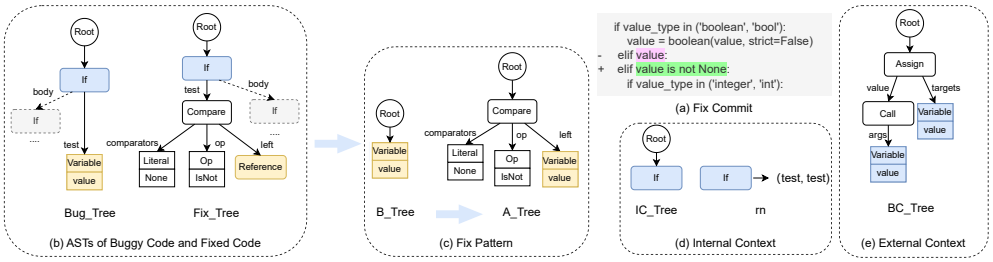


Figure 5.2: An example of fix parsing process on the fix commit ansible:075c6e.

We define external contexts to provide extra location information when B_Tree in the fix pattern and the internal context are both empty. This usually happens when the fix is about adding a new statement and does not modify existing buggy code. The *fix template* is a combination of three components, including the fix pattern, internal context and external context, defined as below.

Definition 5.3.5. (Fix Template) A fix template is a triple (P, IC, EC), where P ($P \neq \emptyset$) is the fix pattern, IC is the internal context, and EC is the external context.

We classify fix templates into four categories based on the fix patterns P:

- **Add**: $B\_Tree = \emptyset \wedge A\_Tree \neq \emptyset$
- **Remove**: $B\_Tree \neq \emptyset \wedge A\_Tree = \emptyset$
- **Insert**: $B\_Tree \neq \emptyset \wedge A\_Tree \neq \emptyset \wedge B\_Tree \subset A\_Tree$
- **Replace**: $B\_Tree \neq \emptyset \wedge A\_Tree \neq \emptyset \wedge B\_Tree \not\subseteq A\_Tree$

Note that there could be more fine-grained classifications under the *Replace* category, such as shuffling the order of statements. However, we find that except for the *Insert* category, these cases are really rare (less than 10 cases in the dataset), so we just adopt the general *Replace* category.

Fix Parsing

In the fix parsing process, TypeFix parses all type error fixes into specific fix templates, with an example illustrated in Fig. 5.2.

Parsing Fix Patterns and Internal Contexts. Given a fix commit, TypeFix first extracts the line information of all added and deleted statements, and then walks through the ASTs of buggy code and fixed code to build template trees. To handle edits at different levels, TypeFix locates the deepest statement-level AST nodes that contain the modified lines, and extracts the corresponding sub-trees in buggy code and fixed code as Bug_Tree and Fix_Tree, respectively. For example, in the fix commit shown in Fig. 5.2(a), TypeFix locates the *If* nodes in the ASTs of buggy code and fixed code, since it is the deepest statement-level AST node containing the edits about variable *value*. Fig. 5.2(b) illustrates the extracted sub-trees as Bug_Tree and Fix_Tree. TypeFix then prunes the same sub-trees shared by Bug_Tree and Fix_Tree to leave only the changed part. For example, the bodies of *If* nodes (grey nodes) are pruned and only the conditions remain in Fig. 5.2(b).

As the pruned Bug_Tree and Fix_Tree contain only the edit, TypeFix can check whether the edit is at the expression level or the statement level. If Bug_Tree and Fix_Tree share the same root node, TypeFix determines that the edit does not rewrite the entire statement and thus it is at the expression level. Otherwise, TypeFix can determine that the edit is at the statement level. For instance, Bug_Tree and Fix_Tree in Fig. 5.2(b) have the same root node If (blue nodes), so the edit is at the expression level. For statement-level edits, the internal context is empty. For expression-level edits, TypeFix creates the internal context by extracting the same nodes shared by Bug_Tree and Fix_Tree to form a new template tree IC_Tree, and subtracts IC_Tree from Bug_Tree and Fix_Tree to build B_Tree and A_Tree in the fix pattern. The relations of

edges that connect IC_Tree in the internal context and B_Tree and A_Tree in the fix pattern are also recorded in the internal context. In the example of Fig. 5.2, the *If* node is extracted as IC_tree in the internal context in Fig. 5.2(d), and the final B_Tree and A_Tree constitute fix pattern in Fig. 5.2(c).

Parsing External Contexts. TypeFix identifies statements that locate outside the scope of the internal context but have direct data dependencies with the fix pattern as external contexts. Specifically, it extracts statements that share the same variables with fix patterns before and after the internal context to build BC_Tree and AC_Tree, respectively. To simplify the fix template abstraction process, TypeFix also prunes the sub-trees in BC_Tree and AC_Tree that do not contain shared variables. For example, in Fig. 5.2, TypeFix identifies the statement *value = boolean(value, strict=False)* because it contains the same variable *value* used in the fix pattern. TypeFix builds a template tree BC_Tree based on this statement and prunes irrelevant sub-trees such as *strict=False*. Since there is no statement after internal contexts that shares the same variables with fix pattern in Fig. 5.2(c), AC_Tree is left empty. The final parsed external context is shown in Fig. 5.2(e).

Fix Template Mining

In the fix template mining process, TypeFix abstracts and merges the specific fix templates into general fix templates via hierarchical clustering. The rationale of template mining is to ensure the least loss of domain knowledge in fix templates. Based on this rationale, TypeFix abstracts or merges the two most similar fix templates each time, and organizes specific to general fix templates as clustering trees. To measure the similarity between two fix templates, we define two kinds of similarity metrics: *value distance* and *structural distance*. The *structural distance* measures the ratio of nodes in two template trees that

have the same type regardless of values (type matching), while the *value distance* measures the ratio of nodes in two template trees that have the same types and values (value matching).

Definition 5.3.6. (Fix Pattern Distances) The value distance d_p and structural distance sd_p between two template trees in fix patterns are defined as

$$d_p(t_1, t_2) = 1 - \frac{VM_p(t_1.rt, t_2.rt)}{\text{Num}(t_1) + \text{Num}(t_2)}$$

$$sd_p(t_1, t_2) = 1 - \frac{TM_p(t_1.rt, t_2.rt)}{\text{Num}(t_1) + \text{Num}(t_2)},$$

where $Num(t)$ indicates the number of nodes in the template tree t, and ValueMatch VM_p and TypeMatch TM_p are defined as

$$VM_p(n_1, n_2) = \begin{cases} 0 & n_1 \neq n_2 \\ 2 + \sum_{i \in \text{child}(n_1, n_2)} VM_p(n_1^i, n_2^i) & \text{otherwise} \end{cases}$$

$$TM_p(n_1, n_2) = \begin{cases} 0 & n_1.t \neq n_2.t \\ 2 + \sum_{i \in \text{child}(n_1, n_2)} TM_p(n_1^i, n_2^i) & \text{otherwise} \end{cases}$$

Definition 5.3.7. (Context Distances) The value distance d_c and structural distance sd_c between two template trees in contexts are defined as

$$d_c(t_1, t_2) = 1 - \frac{\text{MAX}(\text{LeafNode}(t_1), \text{LeafNode}(t_2), VM_c)}{\text{Num}(t_1) + \text{Num}(t_2)}$$

$$sd_c(t_1, t_2) = 1 - \frac{\text{MAX}(\text{LeafNode}(t_1), \text{LeafNode}(t_2), TM_c)}{\text{Num}(t_1) + \text{Num}(t_2)},$$

where MAX(a, b, c) pairs the elements in a and b, and finds the highest similarity c achieved by the pairs, and returns the number of pairs, $Num(t)$ indicates the number of nodes in the template tree t, and $LeafNode(t)$ returns the leaf node set of a template tree t. The ValueMatch VM_c and TypeMatch TM_c are defined as

$$VM_c(n_1, n_2) = \begin{cases} 0 & n_1 \neq n_2 \\ 2 + VM_c(n_1.parent, n_2.parent) & \text{otherwise} \end{cases}$$

$$TM_c(n_1, n_2) = \begin{cases} 0 & n_1.t \neq n_2.t \\ 2 + TM_c(n_1.parent, n_2.parent) & \text{otherwise} \end{cases}$$

To calculate the distances of fix patterns, we adopt a top-down methodology. We start with the root node and require two nodes to be type-matching or value-matching before we compare their child nodes. To calculate the distances of contexts, we adopt a bottom-up methodology. We start with the leaf nodes and require two nodes to be type-matching or value-matching before we compare their parent nodes. Such a difference is caused by the functionality of fix patterns and contexts. The template trees in the fix patterns are used to generate patch code, so based on the definition of ASTs the children nodes are meaningful only if their parent nodes are determined. On the contrary, the template trees in the contexts are used to match the locations that fix patterns should apply instead of generating code, so the children nodes contain more specific location information than the parent nodes. For example, in Fig. 5.2(d), even if we remove the node *Assign*, it can still match the original statement through *Call*, but if we remove the node *Variable*, it can match more general statements that have no direct data dependency with fix pattern in Fig. 5.2(b).

Template Abstraction. TypeFix does not abstract the whole fix template, instead, it abstracts one component, i.e., fix pattern, internal context or external context, each time. TypeFix abstracts the two similar components through a process named *Abstract*. Fig. 5.3 and Fig. 5.4 formally present the processes of *Abstract* on fix patterns and contexts, respectively.

The abstraction of fix patterns and contexts follows the aforementioned top-down and bottom-up methodology, respectively. Generally, there could be four cases when abstracting the template node a and b from two similar template trees:

- **Same Node**: a and b are exactly the same, and they can be reserved for

$$
\begin{aligned}
&\textbf{Abstract}(n_1(C^1_{r_1},...,C^1_{r_m}), n_2(C^2_{r_1},...,C^2_{r_n})) = \\
&\begin{cases}
n_1(O_{r_1},...,O_{r_k}) & \textbf{if } n_1 = n_2 \\
& \textbf{where } k = \min(m,n), \\
& p = \min(\text{len}(C^1_{r_i}), \text{len}(C^2_{r_i})),\ O_{r_i} = \{O^1_{r_i},...,O^p_{r_i}\}, \\
& O^j_{r_i} = \textbf{Abstract}(C^{1j}_{r_i}, C^{2j}_{r_i})\ \forall j \in [1,p] \\
& \text{(Same Node)} \\
o(O_{r_1},...,O_{r_k}) & \textbf{if } n_1.v \neq n_2.v \wedge n_1.t = n_2.t \wedge n_1.bt = n_2.bt \\
& \textbf{where } o.v = ABS,\ o.t = n_1.t,\ o.bt = n_1.bt, \\
& k = \min(m,n),\ p = \min(\text{len}(C^1_{r_i}), \text{len}(C^2_{r_i})), \\
& O_{r_i} = \{O^1_{r_i},...,O^p_{r_i}\}, \\
& O^j_{r_i} = \textbf{Abstract}(C^{1j}_{r_i}, C^{2j}_{r_i})\ \forall j \in [1,p] \\
& \text{(Value Abstraction)} \\
o & \textbf{if } n_1.t \neq n_2.t \wedge n_1.bt = n_2.bt \\
& \textbf{where } o.v = ABS,\ o.t = n_1.bt,\ o.bt = n_1.bt \\
& \text{(Type Abstraction)} \\
\emptyset & \textbf{otherwise} \qquad \text{(Node Removal)}
\end{cases} \\
&\qquad \textbf{where } n_1, n_2 \in A\_Tree.N \textbf{ or } n_1, n_2 \in B\_Tree.N, C^j_{r_i} = \{C^{jt}_{r_i}\} \\
&\textbf{s.t. } \text{Edge}(n^j, C^{jt}_{r_i}, r_i) \in A\_Tree.E \textbf{ or } \text{Edge}(n^j, C^{jt}_{r_i}, r_i) \in B\_Tree.E
\end{aligned}
$$

$$
\begin{aligned}
\textbf{Abstract}(P(a_1,b_1), P(a_2,b_2)) &= P(a,b) \\
\textbf{where } a.rt &= \textbf{Abstract}(a_1.rt, a_2.rt) \\
b.rt &= \textbf{Abstract}(b_1.rt, b_2.rt)
\end{aligned}
$$

Figure 5.3: The process of Abstraction for fix patterns.

the generalized fix template.

- **Value Abstraction**: a and b have the same types but different values. TypeFix creates a node with the same type and set the value as a special ABS token to indicate a hole.
- **Type Abstraction**: a and b have the same base types but different types and values. TypeFix creates a node with the same base type, and sets the type and value as a special ABS token to indicate a hole.
- **Node Removal**: a and b have no common attributes. TypeFix directly

$$\textbf{Abstract}(n_1(P^1_{r_1}), n_2(P^2_{r_2})) =$$

$$\begin{cases} n_1(O_r) & \textbf{if } n_1 = n_2 \\ & \textbf{where } O_r = \textbf{Abstract}(P^1_{r_1}, P^2_{r_2}) \text{ if } r_1 = r_2, \\ & O_r = \emptyset \text{ if } r_1 \neq r_2 \qquad \text{(Same Node)} \\ o(O_r) & \textbf{if } n_1.v \neq n_2.v \wedge n_1.t = n_2.t \wedge n_1.bt = n_2.bt \\ & \textbf{where } o.v = ABS,\ o.t = n_1.t,\ o.bt = n_1.bt, \\ & O_r = \textbf{Abstract}(P^1_{r_1}, P^2_{r_2}) \text{ if } r_1 = r_2, \\ & O_r = \emptyset \text{ if } r_1 \neq r_2 \quad \text{(Value Abstraction)} \\ o & \textbf{if } n_1.t \neq n_2.t \wedge n_1.bt = n_2.bt \\ & \textbf{where } o.v = ABS,\ o.t = n_1.bt,\ o.bt = n_1.bt \\ & \text{(Type Abstraction)} \\ \emptyset & \textbf{otherwise} \qquad \text{(Node Removal)} \end{cases}$$

$$\textbf{where } n_1, n_2 \in IC\_Tree.N \textbf{ or } n_1, n_2 \in BC\_Tree.N$$
$$\textbf{or } n_1, n_2 \in AC\_Tree.N, \text{Edge}(n^j, P^j_{r_j}, r_j) \in IC\_Tree.E$$
$$\textbf{or } \text{Edge}(n^j, P^j_{r_j}, r_j) \in BC\_Tree.E \textbf{ or } \text{Edge}(n^j, P^j_{r_j}, r_j) \in AC\_Tree.E$$

$$\textbf{Abstract}(IC(a_1), IC(a_2), Pairs(c)) = IC(a)$$
$$\textbf{where } \text{LeafNode}(a) = \{\textbf{Abstract}(p^i_1, p^i_2)\}\ \forall p^i \in c$$

$$\textbf{Abstract}(EC(a_1, b_1), EC(a_2, b_2), Pairs(c_1, c_2)) = EC(a, b)$$
$$\textbf{where } \text{LeafNode}(a) = \{\textbf{Abstract}(p^i_1, p^i_2)\}\ \forall p^i \in c_1$$
$$\text{LeafNode}(b) = \{\textbf{Abstract}(p^i_1, p^i_2)\}\ \forall p^i \in c_2$$

Figure 5.4: The process of Abstraction for both internal context and external context.

removes the two nodes.

TypeFix also prunes all child nodes in Type Abstraction and Node Removal, because the change of types for an AST node disables the functionality of its original child nodes.

Mining Fix Templates via Hierarchical Clustering.

With the above-mentioned similarity metrics and abstraction processes, TypeFix selects similar fix templates and merges them via hierarchical clustering to build clustering trees. We give the definition of the clustering tree as follows.

Definition 5.3.8. (Clustering Tree) A clustering tree is a tree (T, E, rt) with fix template set T, edges E and root fix template $rt \in T$. An edge is a pair (t, t') where fix template t is the parent of fix template t', indicating that t is directly abstracted from t'.

To ensure the least loss of domain knowledge, TypeFix follows two strategies in the mining process. First, TypeFix follows the priority order of "external context > internal context > fix pattern" when selecting component pairs for abstraction, ensuring that abstraction of fix patterns happens only if no external context pairs and internal context pairs can be abstracted. Second, TypeFix prefers value abstraction to type abstraction, so it prioritizes component pairs, i.e., fix pattern pairs, internal context pairs or external context pairs, with a structural distance at 0.

Algorithm 3 Fix Template Mining

Input: A set of parsed specific fix templates, T

Output: Mined fix templates, CT

1: $CT \leftarrow T$
2: **while** isChanged(CT) **do**
3: $D_p, D_{IC}, D_{EC} \leftarrow$ CalculateValueDistances(CT)
4: $SD_p, SD_{IC}, SD_{EC} \leftarrow$ CalculateStructuralDistances(CT)
5: $CT \leftarrow$ Deduplicate(CT)
6: $clusters \leftarrow$ selectClusters(CT, D_P, D_{IC}, D_{EC}, SD_{EC})
7: **for** $c \in clusters$ **do** ▷ Handle external contexts
8: $t_1, t_2 \leftarrow$ argmin(c, D_{EC}); $nt \leftarrow t_1$
9: $pairs \leftarrow$ getPairs($D_{EC}(t_1, t_2)$)
10: $nt.EC \leftarrow$ Abstract($t_1.EC, t_2.EC, pairs$, Context)
11: $CT \leftarrow CT - \{t_1, t_2\} + \{nt\}$
12: $t_1.parent, t_2.parent \leftarrow nt$
13: **end for**

14: continue if isChanged(CT)
15: $clusters \leftarrow$ selectClusters(CT, D_P, D_{IC}, SD_{IC})
16: //Handle internal contexts
17: **for** $c \in clusters$ **do**
18: $t_1, t_2 \leftarrow$ argmin(c, D_{EC}); $nt_1 \leftarrow t_1$; $nt_2 \leftarrow t_2$
19: $pairs \leftarrow$ getPairs($D_{IC}(t_1, t_2)$)
20: $nt_1.IC, nt_2.IC \leftarrow$ Abstract($t_1.IC, t_2.IC, pairs$, Context)
21: $CT \leftarrow CT - \{t_1, t_2\} + \{nt_1, nt_2\}$
22: $t_1.parent \leftarrow nt_1$; $t_2.parent \leftarrow nt_2$
23: **end for**
24: continue if isChanged(CT)
25: $clusters \leftarrow$ selectClusters(CT, D_P, SD_P)
26: **for** $c \in clusters$ **do** ▷ Handle fix patterns
27: $t_1, t_2 \leftarrow$ argmin(c, D_P); $nt_1 \leftarrow t_1$; $nt_2 \leftarrow t_2$
28: $nt_1.P, nt_2.P \leftarrow$ Abstract($t_1.P, t_2.P$, Pattern)
29: $CT \leftarrow CT - \{t_1, t_2\} + \{nt_1, nt_2\}$
30: $t_1.parent \leftarrow nt_1$; $t_2.parent \leftarrow nt_2$
31: **end for**
32: **end while**

We present the hierarchical clustering algorithm of TypeFix in Alg. 3. At the beginning of each iteration, TypeFix calculates the distances of three components for every two fix templates (lines 3∼4). TypeFix then removes duplicated fix templates. Based on the calculated distances, TypeFix first handles external context pairs (lines 7 $\sim$ 13), then internal context pairs (lines 16 $\sim$ 22), and finally fix patterns (lines 25 $\sim$ 30). If any abstraction or merge happens, the current iteration will be terminated and a new iteration will begin (lines 14 and 23).

When handling external context pairs, TypeFix groups the fix templates with the same internal contexts and fix patterns into different clusters (line 6). When handling internal context pairs, TypeFix groups the fix templates with the same fix patterns into different clusters (line 15). When handling fix patterns, all fix templates are grouped into one single cluster (line 24). This ensures that only fix templates in the same cluster can be abstracted and merged into more general fix templates under the priority order. For each cluster, TypeFix selects the certain components with the lowest distance in two fix templates (lines 8, 17, 26), and abstracts them into more general components in each iteration (lines 10, 19, 27). The selection has two stages. In the first stage, only components with a structural distance of 0 in the fix templates are considered to prioritize value abstraction. In the second stage, when no such component exists, the rest components are considered. Note that there could be trivial abstractions such as removing all nodes for a template tree so that an empty template tree can represent any code. As empty fix patterns provide no domain knowledge for patch generation, TypeFix does not select two fix patterns whose distance and structural distance are both at 1 in the fix templates for further abstraction.

At the end of each iteration, the new fix templates are included in the set, and the old fix templates are removed from the set (lines 11, 20, 28). The relationships between the new fix templates and the old fix templates are recorded in the clustering tree (lines 12, 21, 29). The merge of fix templates happens only when handling external contexts since the fix patterns and internal contexts of fix templates are already required to be the same at this time. TypeFix also records the number of instances each fix template represent in the training set to facilitate fix template ranking in the next phase. When the template mining process completes, clustering trees that contain specific to general fix templates are generated, facilitating the next patch generation phase.

5.3.3 Patch Generation Phase

The patch generation phase of TypeFix mainly contains two processes: fix template matching and prompt-based patch generation. The fix template matching process aims to select and rank appropriate fix templates that could be applied to the buggy program. The prompt-based patch generation process aims to generate candidate patches by applying selected fix templates to generate code prompts and invoking code pre-trained models for mask prediction.

Fix Template Matching

In the fix template matching process, TypeFix selects matched fix templates on clustering trees via Breadth-First Search (BFS) and then ranks fix templates with frequency and abstraction ratio.

Selecting Fix Templates. Given a buggy program, TypeFix parses the bug lines into a template Tree Bug_Tree, and the contexts before and after the bug lines into template trees $BBug\_Tree$ and $ABug\_Tree$, respectively. TypeFix compares the triple (Bug_Tree, $BBug\_Tree$, $ABug\_Tree$) with fix templates in the clustering trees to find the appropriate fix templates. We define the following rules to check whether a buggy program matches a fix template.

Definition 5.3.9. (Template Node Match) For two template nodes a and b, a matches b if $a.value$ matches $b.value$ and $(a.t, a.bt)$ matches $(b.t, b.bt)$. $a.value$ matches $b.value$ if $b.value = ABS \vee a.value = b.value$. $(a.t, a.bt)$ matches $(b.t, b.bt)$ if $a.bt = b.bt \wedge (a.bt = b.t \vee a.t = b.t)$.

Definition 5.3.10. (Template Tree Match) For two template trees A and B, A matches B if there is a node $a \in A.N$ where a matches to $B.rt$ and there exists node maps $\{ac_1 \rightarrow bc_1, ..., ac_n \rightarrow bc_n\}$ where ac_i matches bc_i, $\{ac_1, ..., ac_n\} \subseteq a.children$, $ac_{i+1}.id > ac_i.id$, and $\{bc_1, ..., bc_n\} = B.rt.children$. A always matches

B if $B = \emptyset$.

Definition 5.3.11. (Fix Template Match) For a buggy program (Bug_Tree, $BBug\_Tree$, $ABug\_Tree$) and a fix template (P, IC, EC), the buggy program matches the fix template if $BBug\_Tree$ matches $EC.BC\_Tree$, $ABug\_Tree$ matches $EC.A\_Tree$ and Bug_Tree matches Concat($IC.IC\_Tree$, $P.B\_Tree$,$IC.rn$), where Concat(a, b, rn) indicates concatenating template tree b to template tree a with edge (n, $b.rt$, $rn[n].br$).

With the above rules, TypeFix starts with the root fix template (most general fix templates) of each clustering tree and walks through the clustering tree via bread-first search (BFS) until it finds the deepest fix template (most specific fix templates) matched by the buggy program. These fix templates are collected to be ranked in the next step.

Ranking Fix Templates. TypeFix ranks the fix templates before applying them to the buggy program. To provide the most domain knowledge for pre-trained models in the patch generation process, TypeFix utilizes a two-step strategy to prioritize fix templates.

TypeFix groups the fix templates with the same concatenated template tree of IC_Tree and B_Tree. These fix templates provide different fix solutions for the same buggy pattern. TypeFix ranks fix templates in one group based on the number of training instances they represent because a larger number indicates that the fix template is used more frequently on the given buggy program. TypeFix then ranks the groups based on the abstraction ratio of A_Tree of the first fix template in each group. The abstraction ratio of a template tree is defined by the ratio of nodes whose values or types are ABS tokens. A higher abstraction ratio of A_Tree indicates less domain knowledge associated, so that code pre-trained models need to predict more information before they can generate complete candidate patches. For example, an abstraction ratio of 1.0 indicates

the fix template actually is a huge hole and there is no domain knowledge assisting the generation of patches. Therefore, TypeFix prioritizes the groups with a lower abstraction ratio to include more domain knowledge in the patch generation process.

Prompt-based Patch Generation

In the process, TypeFix applies ranked fix templates on the buggy program and generates code prompts. CodeT5 model is then invoked to fill the masks in code prompts and generate candidate patches.

Applying Fix Templates. For each selected fix template, TypeFix completes the A_Tree in its fix pattern by adding dummy AST nodes, i.e., AST nodes with values of ABS tokens, as placeholders, because some child AST nodes are removed in the fix template mining process. TypeFix then replaces the sub-tree AST of the buggy program that matches B_Tree with the completed A_Tree, and converts the modified AST of the buggy program to code prompts. Code prompts are source code that contains ABS tokens as masks to be predicted by code pre-trained models.

Generating Patches. Most code pre-trained models are trained to predict masks in source code, thus they can naturally be used to predict the value of ABS tokens in the code prompts. In this chapter, we choose CodeT5 [217] as the code pre-trained model in the patch generation process, since it is specially designed for the code generation task [217]. When generating patches, TypeFix replaces the ABS tokens in the code prompt with ordered mask tokens used in CodeT5, e.g., <extra_id_0>, ..., <extra_id_99>. TypeFix then invokes CodeT5 to predict tokens for each mask. The predicted values for the masks are filled into the code prompts to generate candidate patches.

Validating Patches. TypeFix adopts the classic generate-and-validate

methodology in patch generation. For the generated patches, TypeFix first filters out those with syntax errors, and then runs the test suite on each patch to find plausible patches, i.e., those can successfully pass all test cases. Plausible patches are further examined by the authors to identify correct patches, i.e., those are semantically identical to the developer patch when ignoring I/O side effects such as messages in *print* statements.

5.4 Experiment Setup

5.4.1 Dataset

Training Set. Following previous work [143], we build a dataset for the fix template mining process of TypeFix and the training of baselines. We collect 8,722 merged pull requests from GitHub that contain the term "fix type error". We extract the fixes from the commits in collected pull requests. We remove the overlong commits that contain more than 50 lines of modified code. Finally, we get 10,981 fixes to form the training set.

Benchmarks. Following previous work [143], we use two benchmarks BugsInPy [224] and TypeBugs [143]. The two benchmarks initially separate type errors by commits, but we find that a single commit can also involve more than one type errors in different locations. To avoid the correct fix of one type error being hidden by another type error, we further split the commits that contain two or more type errors into multiple ones. We also remove the duplicated type errors, i.e., those that have the same commit signatures, in two benchmarks. Finally, we get 54 type errors from BugsInPy and 109 type errors from TypeBugs.

5.4.2 Baselines

We compare TypeFix with the following four baselines.

PyTER. PyTER [143] is a rule-based APR approach designed for repairing Python type errors. It has nine pre-defined templates and several rules to synthesize templates to generate candidate patches.

AlphaRepair. AlphaRepair [228] is the state-of-the-art prompt-based approach for general-purpose APR. It masks tokens in the buggy code based on some general prompt templates and invokes code pre-trained models to generate patches.

CoCoNuT. CoCoNuT [112] is an NMT-based approach for general-purpose APR. It translates the buggy code into candidate patches.

Codex. Codex [19] is a large GPT model fine-tuned on publicly available code from GitHub. It is designed by OpenAI and used to power GitHub Copilot [122] service.

5.4.3 Metrics

We adopt the commonly used **Correct** and **Plausible** metrics in previous work [86, 112, 143, 228] to evaluate the performance of TypeFix on repairing type errors. Besides, we add a new metric named **Template Coverage** to evaluate the number of developer patches covered by the fix templates mined by TypeFix and pre-defined ones from PyTER. **Template Coverage** is defined by the ratio of bugs whose developer patch matches a fix template of an approach.

5.4.4 Implementation

The entire framework of TypeFix is implemented using Python, which contains more than 10,000 lines of code. We adopt the CodeT5-base [185] model to

predict the masks in code prompts and generate candidate patches. For PyTER, AlphaRepair and CoCoNuT, we directly use the replication packages released by the authors and re-implement them on our task. We train CoCoNuT with its original training set and the training set we collected to adapt it to fix Python type errors. Since Codex is not publicly available, we use the public API [146] of engine *code-davinci-002* provided by OpenAI to query it with prompts. We use a similar prompt from previous work [226]. The only difference is that we use three examples instead of two at the beginning of the prompt and only mask the buggy line to maximize the performance of Codex. We make all other settings consistent with previous work [86, 112, 143, 226, 228]. All experiments are conducted on a Linux machine (Ubuntu 20.04) with two Intel Xeon@2.20GHZ CPUs, one NVIDIA A100-SXM4-40GB GPU and 256GB RAM.

5.5 Evaluation

In this section, we evaluate the performance of TypeFix on the following three research questions:

- **RQ1**: How effective is TypeFix to fix type errors?
- **RQ2**: How capable is TypeFix to mine fix templates from existing bug fixes?
- **RQ3**: When does TypeFix fail to fix type errors?

5.5.1 RQ1: Effectiveness of TypeFix

To evaluate the effectiveness of TypeFix on repairing type errors, we compare TypeFix with state-of-the-art rule-based APR approaches and learning-

based APR approaches. Table 5.1 presents the performance of TypeFix along with baseline approaches on two benchmarks TypeBugs and BugsInPy.

Table 5.1: Evaluation results of TypeFix compared with three baselines. Results are presented in the Correct/Plausible format. Fix rate is the ratio of correct patches.

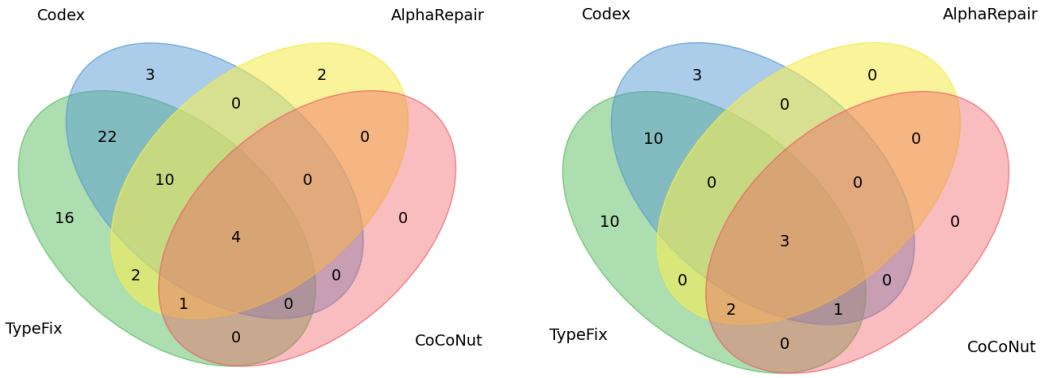
TypeBugs						
Project	**#B**	**TypeFix**	**PyTER**	**Codex**	**AlphaRepair**	**CoCoNuT**
airflow	14	9/9	4/4	7/7	1/6	0/4
beets	1	0/0	0/1	0/0	0/0	0/0
core	9	7/7	5/7	4/5	4/4	2/3
kivy	1	0/0	0/1	0/0	0/1	0/1
luigi	2	0/2	0/0	0/2	1/2	0/0
numpy	3	0/3	0/2	0/1	0/2	0/0
pandas	48	21/32	17/27	18/19	11/22	3/10
rasa	2	2/2	0/0	2/2	0/0	0/0
requests	4	4/4	4/4	2/2	0/1	0/0
rich	4	2/3	0/1	1/1	0/0	0/0
salt	8	5/8	5/5	4/5	1/5	0/2
sanic	2	0/0	2/2	0/0	0/0	0/0
scikit-learn	7	2/3	2/3	1/2	0/0	0/0
tornado	1	0/0	1/1	0/0	0/0	0/0
Zappa	3	3/3	1/1	0/0	1/3	0/1
Total	109	**55/76**	41/59	39/46	19/46	5/21
Fix Rate (%)	-	**50.5**	37.6	35.8	17.4	4.6
BugsInPy						
Project	**#B**	**TypeFix**	**PyTER**	**Codex**	**AlphaRepair**	**CoCoNuT**
ansible	1	0/0	0/0	0/0	0/0	0/0
fastapi	1	1/1	0/0	1/1	0/0	0/0
keras	7	4/6	1/1	0/3	0/4	0/4
luigi	7	4/5	3/5	3/3	0/0	0/0
pandas	19	4/13	4/6	2/6	3/10	3/8
scrapy	12	10/11	5/7	10/12	1/4	2/4
spacy	1	0/1	0/1	0/0	0/1	0/1
tornado	2	1/1	1/1	1/1	0/1	0/1
youtube-dl	4	2/3	1/1	0/2	1/1	1/1
Total	54	**26/41**	15/22	17/28	5/21	6/19
Fix Rate (%)	-	**48.1**	27.8	31.5	9.3	11.1

Table 5.2: Comparison of the number of unique type error fixes and template coverage between TypeFix and PyTER.

Approach	TypeBugs		BugsInPy	
	#Unique	Coverage	#Unique	Coverage
TypeFix	24	83 (76.1%)	16	40 (74.1%)
PyTER	10	46 (42.2%)	5	18 (33.3%)

Comparison with Rule-based Approach. As can be seen in Table 5.1, TypeFix can successfully fix 55 type errors in TypeBugs and 26 type errors in BugsInPy, outperforming rule-based approach PyTER by 14 and 11 type errors, respectively. We attribute the improvement of TypeFix to the higher coverage of fix templates mined from existing type error fixes and the generated domain-aware code prompts. Furthermore, we analyze the unique type errors that TypeFix and PyTER can fix in two benchmarks and present the results in Table 5.2. We find that TypeFix obtains 24 and 16 unique type error fixes in TypeBugs and BugsInPy, respectively, while PyTER only obtains 10 and 5 unique type error fixes in TypeBugs and BugsInPy, respectively. This further demonstrates the effectiveness of TypeFix when compared with PyTER.

Comparison with Learning-based Approaches. From Table 5.1 we can see that TypeFix, Codex and AlphaRepair generally perform much better than CoCoNuT, indicating the superior performance of prompt-based approaches. When comparing TypeFix with AlphaRepair which adopts general domain-unaware prompt templates, we find that TypeFix achieves a $1\times \sim 4\times$ larger fix rate than AlphaRepair. This indicates that general domain-unaware prompt templates such as randomly replacing several tokens in code can hardly handle complicated type errors. Compared with the most advanced code pre-trained model Codex, TypeFix still obtains a significant improvement by fixing



(a) TypeBugs (b) BugsInPy

Figure 5.5: Venn diagram of correct patches provided by learning-based APR approaches.

16 and 9 more type errors than Codex in TypeBugs and BugsInPy, respectively. This improvement further demonstrates the importance of domain knowledge for repairing type errors, even though Codex has a much larger parameter size (12B) than that (220M) of the CodeT5 model utilized by TypeFix.

In addition to the total number of type errors fixed by each approach, we further evaluate the number of unique type error fixes. Fig. 5.5 presents the unique type errors that TypeFix and three learning-based approaches can correctly fix in the format of Venn diagrams. We observe that TypeFix obtains 16 and 10 unique type error fixes in TypeBugs and BugsInPy, respectively, while other approaches only obtain $0 \sim 3$ unique bug fixes in two benchmarks. This indicates that the contribution of domain-aware fix templates cannot be replaced by the combination of existing learning-based approaches.

Answer to RQ1: TypeFix successfully fixes 55 and 26 bugs in two benchmarks, outperforming state-of-the-art approaches by at least 14 bugs and 9

Table 5.3: Statistics of fix templates mining in TypeFix.

Category	#Instances	#Clustering Trees (>1/>5)	Mining Time/s
Add	2,656	150/27	2,819
Remove	570	10/5	59
Insert	1,648	350/47	659
Replace	6,107	184/70	32,621

bugs, respectively. Meanwhile, TypeFix obtains the most unique type error fixes in two benchmarks.

5.5.2 RQ2: Capability of TypeFix to Mine Fix Templates

To comprehensively investigate the capability of TypeFix to mine fix templates, we focus on the performance of TypeFix in template mining and the usefulness of fix templates mined by TypeFix.

Table 5.3 presents the performance of TypeFix in fix template mining process. Starting with thousands of existing bug fixes, TypeFix can mine $10 \sim 350$ clustering trees. After discarding the clustering trees with occurrence frequency lower than a threshold (5 in this chapter), TypeFix finally gets $5 \sim 70$ clustering trees. The mining process generally takes shorter than one minute to at most nine hours. Table 5.2 presents the template coverage achieved by TypeFix and PyTER. From it we can observe that fix templates mined by TypeFix can cover about 75% of type errors in two benchmarks while the manually defined fix templates in PyTER can only cover about $30\% \sim 40\%$ of type errors.

To further study how fix templates mined by TypeFix can help the patch generation process, we conduct an ablation study on fix templates under each category. Following previous study [228], we start with the case that no fix

Table 5.4: Ablation results.

	TypeBugs		BugsInPy	
	#Correct	**#Plausible**	**#Correct**	**#Plausible**
No Template	19	41	6	20
Add	+11	+10	+3	+5
Remove	+1	+1	+0	+0
Replace	+6	+8	+4	+9
Insert	+18	+16	+13	+17
Total	55	76	26	41

template is applied, i.e., the CodeT5 model is asked to generate a completely new line to replace the original buggy line. We then gradually apply fix templates under *Add*, *Remove*, *Replace* and *Insert* categories, and observe the number of new correct and plausible patches, respectively. We show the results in Table 5.4. We can find that all four categories of fix templates contribute to generating correct patches. This demonstrates the contribution of domain knowledge stored in the fix templates. We also note that fix templates under *Insert* and *Add* categories contribute the most. The reason could be attributed to that developers often add guards to guarantee the desired types or directly convert input types into desired types when fixing type errors.

Answer to RQ2: TypeFix achieves a template coverage of about 75% on both benchmarks. Ablation results also demonstrate the usefulness of fix templates mined by TypeFix under each category.

5.5.3 RQ3: Limitations of TypeFix

Our experiments also show the limitations of TypeFix as it cannot fix all type errors in two benchmarks. By analyzing the type errors that TypeFix cannot fix in two benchmarks, we conclude two possible limitations.

The first limitation is that TypeFix cannot always find matched fix templates to the current buggy program. Based on Table 5.2, we can find that even if fix templates mined by TypeFix can cover as many as 75% cases in two benchmarks, there exist a few cases (∼25%) that do not share similar patterns with instances in the training set. An example is illustrated in the first type error of Listing 5.2. To fix this type error, the developer changes a list comprehension into an attribute access, which does not appear in the training set. TypeFix thus cannot find proper fix templates for this type error and fails to fix it. This limitation can be mitigated by adapting TypeFix to new datasets, so that TypeFix can mine new fix templates to improve the template coverage.

```
#Type Error 1: apache/airflow:892d4d
if conf.getboolean('core', 'store_dag_code',\
fallback=False):
-     DagCode.bulk_sync_to_db([dag.fileloc for dag in orm_dag])
+     DagCode.bulk_sync_to_db([orm_dag.fileloc])
#Type Error 2: pandas-dev/pandas:a3e903
elif (is_extension_array_dtype(left) or\
- is_extension_array_dtype(right)):
+ (is_extension_array_dtype(right) and not is_scalar(right))):
        return dispatch_to_extension_op(op, left, right)
```

Listing 5.2: Two type errors that TypeFix fail to fix

The second reason is that the CodeT5 model TypeFix uses sometimes cannot generate the correct patches even if the correct fix template is given. By comparing Table 5.1 and Table 5.2, we can find that fix templates mined by TypeFix

can cover 83 bugs in TypeBugs but only 55 of them are correctly fixed. This indicates the limitations of CodeT5 when generating candidate patches from code prompts. We also show an example as the second type error of Listing 5.2. In this type error, we need to add a new condition as the guards and this fix pattern is commonly used in the wild. However, CodeT5 cannot give *is_scalar* as the new condition and thus TypeFix fails to fix this type error. We believe this limitation can be mitigated by using more advanced code pre-trained models, as the parameter size of CodeT5 is only 220M.

Answer to RQ3: TypeFix sometimes fails to fix type errors due to the limited performance of pre-trained code models and a few cases ($\sim$ 25%) that mined fix templates cannot cover.

5.6 Summary

We propose a domain-aware prompt-based approach named TypeFix for repairing Python type errors. TypeFix improves prompt-based approach by incorporating domain-aware fix templates. TypeFix implements a novel fix template design to handle type errors at different levels, and mines fix templates via a novel hierarchical clustering algorithm. TypeFix incorporates domain knowledge into code prompts by applying fix templates into buggy code and invokes code pre-trained models to generate candidate patches from code prompts. Experiments demonstrate the effectiveness of TypeFix and the usefulness of fix templates mined by TypeFix.

Chapter 6

Evaluation Study for Automatic Third-party API Recommendation Approaches

With the rich support of third-party packages in Python, developers can easily invoke external APIs to avoid implementing similar functionalities in the development of Python software. Given the large volume of external APIs, even the most skilled developers cannot be familiar with all of them. Therefore, it is essential to develop API recommendation approaches to help developers select the most appropriate APIs to avoid potential run-time environment conflicts caused by the absence of the implementations of external APIs in the run-time environment. In this chapter, we evaluate existing API recommendation approaches and aim to identify the challenges of recommending high-quality APIs. The main points of this chapter are as follows. (1) we systematically study both query-based and code-based API recommendation techniques on two large-scale datasets including Java and Python. (2) We build an open-sourced benchmark named APIBench to fairly evaluate query-based and code-based approaches.

(3) We study how different settings can impact the performance of current approaches, including query quality, cross-domain adaptation, etc. (4) We conclude some findings and implications that would be important for future research in API recommendation.

6.1 Introduction

Application Programming Interfaces (APIs) provided by software libraries or frameworks play an important role in modern software development. Almost all programs, even the basic "hello world!" program, include at least one API. However, there are a huge number of APIs from different modules or libraries. For example, the Java standard library [148] provides more than 30,000 APIs. It is therefore infeasible for developers to be familiar with all APIs. To address this problem, many approaches are proposed to recommend APIs based on input queries, which describe the programming task in natural language, or surrounding context, i.e., the code already written by developers.

However, a uniform definition of the current API recommendation task is still absent, making the task hard to be followed by potential researchers. Some studies [15, 90, 135, 177, 181] regard the task as a code completion problem, and recommend any code tokens including APIs. These studies focus on improving the prediction results of all the tokens instead of only APIs. Some studies [60, 76, 109, 170, 172] recommend relative APIs on different levels given natural language queries. Besides, the evaluation results are difficult to be reproduced by future related work. For example, for query-based API recommendation, manual evaluation is generally adopted, so the performance reported by different studies can hardly be aligned. Comparing with widely-used Integrated Development Enviroments (IDEs) or search engines is another commonly adopted yet inconsistent

evaluation strategy in previous research. Therefore, to better facilitate future exploration of the API recommendation task, in this chapter, we summarize the recent related approaches and build a general benchmark named APIBench.

To facilitate the benchmark creation, we group the recent related approaches into two categories according to the task definition: query-based API recommendation and code-based API recommendation:

1) **query-based API recommendation.** Approaches for query-based API recommendation aim at providing related APIs to developers given a query that describes programming requirements in natural language. The approaches can inform developers which API to use for a programming task.

2) **code-based API recommendation.** Approaches for code-based API recommendation aim at predicting the next API given the code surrounding the point of prediction. They can directly improve the efficiency of coding.

Besides the unreproducible evaluation, the two groups of studies face their own challenges. 1) For query-based approaches, high-quality queries play a critical role in accurate recommendation. However, there may exist a knowledge gap between developers and API designers in choosing terms for describing queries or APIs. For example, developers who do not know the term "*heterogeneous list*" in API documents would use other words such as "*list with different types of elements*" in the query. Whether current query reformulation techniques are effective for API recommendation and how effective it is are still remaining unexplored. 2) For code-based approaches, the quality of code before the recommendation point also affects the recommendation performance. Generally, the approaches are evaluated by simulating an actual development, i.e., some parts of a project are removed for imitating a limited context. The APIs to recommend may be located in the front, middle, or back of the code, so exploring the impact of different recommendation points is important for understanding the recom-

mendation capability of existing approaches. Other factors such as whether the APIs are standard or user-defined, lengths of given context, and different domains can also influence the recommendation performance, which have not yet been fully investigated.

To comprehensively understand the above challenges and align the performance of current approaches, we first build a benchmark named APIBench. APIBench is built on Python and Java, and involves two datasets for evaluation, named as APIBench-Q and APIBench-C for query-based and code-based approaches, respectively. APIBench-Q contains 6,563 Java queries and 4,309 Python queries obtained from Stack Overflow and API tutorial websites. APIBench-C contains 1,477 Java projects with 1,229,698 source files and 2,223 Python projects with 414,753 source files obtained from GitHub. Based on APIBench, we study the following research questions:

- **RQ1:** How effective are current query-based and code-based API recommendation approaches?
- **RQ2:** What is the impact of query reformulation techniques on the performance of query-based API recommendation?
- **RQ3:** What is the impact of different data sources on the performance of query-based API recommendations?
- **RQ4:** How well do code-based approaches recommend different kinds of APIs?
- **RQ5:** What is the performance of code-based approaches in handling different contexts?
- **RQ6:** How well do code-based approaches perform in cross-domain scenarios?

APIBench involves the implementation of the related approaches proposed in the recent five years, specifically including five query-based approaches and five code-based approaches. In RQ1, we compare the performances of the approaches in APIBench. To answer RQ2 and RQ3, we apply four popular query reformulation techniques to the queries of APIBench-Q and observe the performance of the query-based approaches given reformulated queries. To answer RQ4 to RQ6, we analyze the APIs in APIBench-C from different aspects and study the performance of code-based approaches under different experimental settings.

Key Findings. Through the large-scale empirical study, we achieve some findings and summarize the key findings as below.

(1) For query-based API recommendation:

- While current approaches make a good progress on class-level recommendation, recommending the exact API methods is still a challenging task.
- Query reformulation techniques, including query expansion and query modification, are quite effective in improving the performance of query-based approaches.
- Adding data sources such as Q&A forums and tutorials that are more similar to real-world queries can significantly improve the performance of current approaches.

(2) For code-based API recommendation:

- Recent deep learning models such as Transformers show superior performance on this task. Meanwhile, current IDEs can achieve competitive performance as recent pattern-based and learning-based approaches. They work far more than just recommending APIs based on alphabet orders.

- Current approaches are effective to recommend APIs from standard libraries and popular third-party libraries, but their performance drops a lot when recommending user-defined or project-specific APIs.
- Approaches trained on one single domain face the problem of cross-domain adaptation. Approaches trained on multiple domains achieve satisfying performance when testing on most single domains, and they even outperform those trained on corresponding single domains.

Based on the findings, we conclude some implications and suggestions that would benefit future research. On the one hand, query-based API recommendation approaches should be built along with query reformulation techniques to handle queries with different qualities. We also encourage future work to leverage different data sources and few-shot learning methods to address the low resource challenge in query-based API recommendation. On the other hand, we suggest future code-based API recommendation approaches focus on improving the performance of recommending user-defined APIs as it is currently the major bottleneck.

Contributions. To sum up, our contribution can be concluded as follows.

- To the best of our knowledge, we are the first to systematically study both query-based and code-based API recommendation techniques on two large-scale datasets including Java and Python.
- We build an open-sourced benchmark named APIBench to fairly evaluate query-based and code-based approaches.
- We study how different settings can impact the performance of current approaches, including query quality, cross domain adaptation, etc.

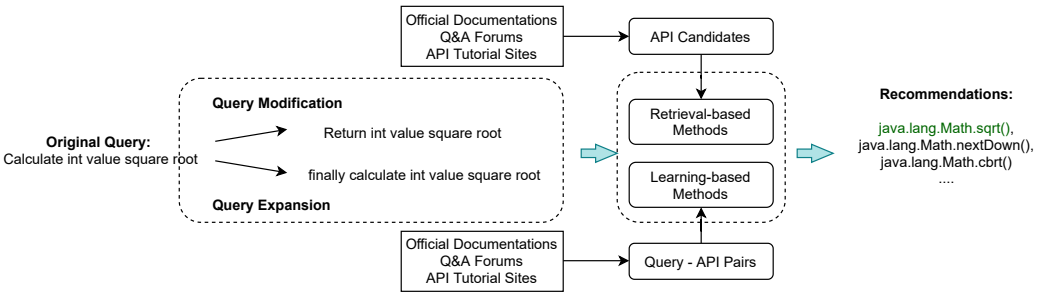


Figure 6.1: The typical query-based API recommendation framework.

- We conclude some findings and implications that would be important for future research in API recommendation.

6.2 Background

In this section, we summarize the query-based approaches and code-based approaches, respectively.

6.2.1 Query-Based API Recommendation Methods

We describe the typical query-based API recommendation process in Figure 6.1. Given a query "*Calculate int value square root*", query reformulation techniques first modify the query as "*return int value square root*" or expand it as "*finally calculate int value square root*". A knowledge base built upon available data sources is also prepared for API candidate selection. Based on the knowledge base, retrieval-based methods or learning-based methods recommend the APIs relevant to the queries.

Query Reformulation Techniques

Input queries can be short in length or vague in semantics. Besides, there may exist a knowledge gap between developers and search engines in query description. For rendering search engines better understand the query semantics, query reformulation is a common pre-processing method. In general, there are two major types of query reformulation approaches:

1) query expansion, which adds extra information to the original queries;

2) query modification, which modifies, replaces or deletes some words in the original queries.

Query expansion. Query expansion aims at identifying important words that are missing in the input queries. The topic is originally stemmed from the field of natural language processing (NLP). For example, the work [111] utilizes word embeddings to map words in the vector space and finds similar words to enrich the queries. For the API recommendation task, since APIs are encapsulated and organized according to classes and modules, class names and module names are important hints for recommendation. Rahman *et al.* [170, 172] propose to use keyword-API class co-occurrence frequencies and keyword-keyword co-occurrence frequencies to build the relationship between words and API classes, and add the suggested API class for query expansion.

Query modification. Query modification aims at mitigating both the lexical gap and knowledge gap between the user queries and descriptions in knowledge base. The lexical gap, such as mis-spelling, can be easily addressed by spelling correction and synonym search, etc. Recent work focuses on how to mitigate the knowledge gap by replacing inappropriate words in queries. Mohammad *et al.* [4] extract important tokens in code, and Sirres *et al.* [191] leverage discussions and code from Stack Overflow posts to build a knowledge base. Cao *et al.* [16] collect query reformulation history from Stack Overflow and propose a

Transformer-based approach to learn how developers change their queries when search engines do not return desired results.

Recommendation with Knowledge Base

Knowledge base. API recommendation approaches generally require a knowledge base that contains all the existing APIs as the search space. There are three primary sources for the knowledge base creation, including: 1) official documentations which contain comprehensive descriptions about the API functionality and structure. 2) Q&A forums, which provide the purposes of APIs and different API usage patterns. Many studies [76, 169] leverage the Q&A pairs from Stack Overflow to select API candidates. 3) Wiki sites, which describe concepts that link different APIs. For example, Liu *et al.* [109] utilizes API concepts from Wikipedia to help build API knowledge graphs.

Retrieval-based methods. Retrieval-based methods retrieve API candidates from the knowledge base and then rank the candidate APIs by calculating the similarities between queries and APIs. For example, Rahman *et al.* [170, 172] utilize the keyword-API occurrence frequencies and API-API occurrence frequencies to find the most relevant APIs. Huang *et al.* [76] first identify the similar posts from Stack Overflow by computing query-documentation similarities and choose the APIs mentioned in posts as candidates. Liu *et al.* [109] build an API knowledge graph to represent relationships between APIs and then calculate the similarities between queries and certain parts of API knowledge graph to rank the APIs.

Learning-based methods. Another type of method is to automatically learn the relationships between queries and APIs based on deep learning techniques. The knowledge base provides query-API pairs as the ground truth. For example, Gu *et al.* [60] formulate the task as a translation problem in which a

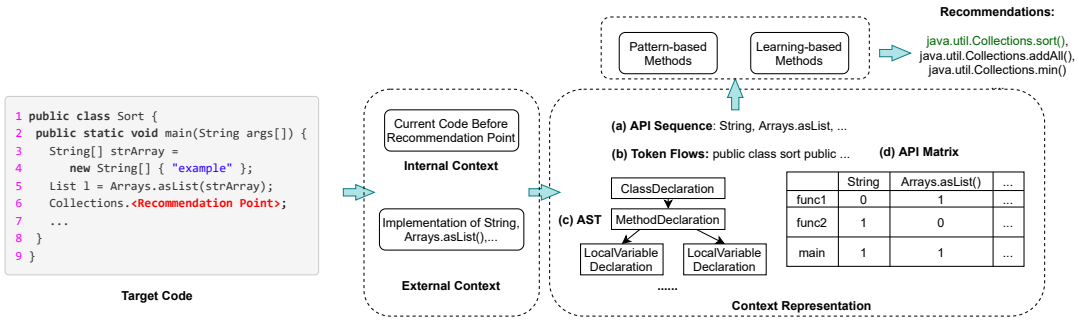


Figure 6.2: The typical code-based API recommendation framework.

model is built to translate word sequences into API sequences. They propose an RNN model with an encoder-decoder structure to implement the translation.

6.2.2 Code-Based API Recommendation Methods

We describe the workflow of code-based API recommendation in Figure 6.2. Given a target code, context representation is an essential step. Based on the extracted context, pattern-based methods or learning-based methods are adopted by previous studies to recommend the next API.

Context for the Target Code

Most code-based API recommendation methods regard the code before the recommendation point as the context. We name such context as *internal context* since it only considers code in the current source code or current function body. For example, Line 1 $\sim$ 6 of the target code in Figure 6.2 belongs to internal context. Xie *et al.* [231] find that replacing external APIs in code (such as *Arrays.asList*() in Figure 6.2) with their implementations can help the identification of common usage patterns. They propose to build a hierarchical context by integrating the implementation out of the current source file. We name the implementation of external APIs as *external context*.

Context Representation

We divide the context representation methods into two types, i.e., pattern-based representation and learning-based representation. **Pattern-based representations** [28, 137, 139, 222, 231] do not consider all the code tokens. Instead, they only identify APIs to build API usage sequences, as shown in Figure 6.2 (a), API matrix, as shown in Figure 6.2 (d), or API dependency graphs to represent the current context. **Learning-based representations** [66, 69, 90, 177, 203] usually represent the context with token flows, as illustrated in Figure 6.2 (b), or other syntax structures such as Abstract Syntax Trees (ASTs), as illustrated in Figure 6.2 (c).

Recommendation Based on Context

Pattern-based methods. API recommendation is inherently a recommendation task, so some studies [28, 139] follow the collaborative filtering (*user-item*) methodology of traditional recommendation systems [179]. As shown in Figure 6.2 (d), they regard the internal context as the *users* and APIs as the *items*. They then calculate the similarities between different *users* to find the most similar API for recommendation. However, the methods do not consider the relationships between APIs. More recent work [222, 231] build API dependency graphs or mines association rules to capture API usage patterns.

Learning-based methods. Hindle *et al.* [69] discover the naturalness of software, rendering it possible to deploy machine learning or deep learning methods on code. Different from pattern-based methods that consider the relationships between API occurrences, learning-based methods regard API as a single code token, and reformulate the code-based API recommendation problem into a *next token prediction* problem. Many statistical language models [136, 175, 177, 203] are proposed to predict the next code token. Besides using the token sequences,

more recent work [66, 90] try to leverage syntax and data flow information for more accurate prediction.

Note that we do not aim to provide a comprehensive summary of all query-based and code-based API recommendation approaches but to choose some representatives to describe the general workflow in this section. For a more comprehensive literature review, we refer the readers to previous surveys and empirical studies [93, 183, 184].

6.3 Methodology

In this section, we introduce the scope of the studied APIs, the preparation of benchmark datasets, and implementation details.

6.3.1 Scope of APIs

To fairly compare the current API recommendation approaches, benchmark datasets should be prepared, during which the scope of studied APIs first needs to be defined. In this work, we focus our evaluation on two popular programming languages, i.e., Python and Java.

For facilitating the analysis of the challenges in API recommendation, we divide all APIs into *standard APIs*, *user-defined APIs*, and *popular third-party APIs*. The *standard APIs* refer to the APIs that are clearly defined and built-in in corresponding programming languages while the *user-defined APIs* are defined and used in projects along with *popular third-party APIs*. Following previous work [76, 109, 170, 172] we evaluate query-based API recommendation methods only on the *standard APIs* since currently *standard APIs* have the most comprehensive documentations and extensive discussions to build the knowledge base. We evaluate code-based API recommendation methods on all three kinds of APIs.

The details of each kind for different programming languages are depicted below.

(1) Standard Java APIs. We chose the version Java 8 for our analysis since it is the most widely-used version in current projects according to the 2020 JVM Ecosystem Report [192]. We collect 34,072 APIs from the Java documentation [148] as *standard APIs*.

(2) Android APIs. We choose APIs from the Android library [57] since Android is the most popular application of Java programs. We collect 11,802 APIs from the official documentation of Android in total.

(3) Standard Python APIs. As Python Software Foundation has stopped the support for Python 2, currently only 6% of developers are still using Python 2, according to the development survey conducted by Jetbrains [82]. Considering that APIs of different versions above 3.0 are similar, we choose the newest version 3.9 to ensure compatibility and collect 5,241 APIs from Python standard library [164] as the *standard APIs*.

(4) Popular Python third-party APIs. Python is well extended by a lot of third-party modules. We choose five widely-used modules with sufficient documentations, including flask [32], django [25], matplotlib [116], pandas [149] and numpy [142]. We collect 215, 700, 4,089, 3,296, and 3,683 APIs from them, respectively.

(5) User-defined APIs. For code-based API recommendation, we regard all the functions defined in current projects as *user-defined APIs*. We do not explicitly collect them as a fixed set because they vary across projects. By inspecting the implementations, we can always identify the user-defined APIs.

6.3.2 Benchmark Datasets

In this section, we describe how we build the benchmark datasets APIBench-Q and APIBench-C.

Table 6.1: Statistics of APIBench-Q. Ori. represent the original queries, Exp. represent the expanded queries produced by query expansion techniques, Mod. represents the modified queries produced by query modification techniques.

PL	Stack Overflow			Tutorial Websites		
	Ori.	Exp.	Mod.	Ori.	Exp.	Mod.
Python	1,925	78,157	100,100	2,384	95,360	123,968
Java	1,320	80,343	68,640	5,243	319, 783	272,636

Creation of APIBench-Q

We build the benchmark dataset APIBench-Q by mining Stack Overflow and tutorial websites. Note that we find that currently there is no query-based API recommendation approach specially designed for Python programs, but we still collect the query benchmark for it to facilitate further research investigation.

Mining Stack Overflow. As one of the most popular Q&A forums for developers, Stack Overflow contains much discussion about the usage of APIs. Stack Overflow is the primary source for building APIBench-Q. We first download all posts from Aug 2008 to Feb 2021 on Stack Overflow (SO) via Stack Exchange Data Dump [30]. Each post is associated with a tag about the related programming language. We filter out the posts not tagged as Java or Python, resulting in 1,756,183 Java posts and 1,661,383 Python posts. Similar to other studies related to Stack Overflow mining [27], [55], we further filter out the posts based on the following rules:

- To increase the quality of the posts, we remove the posts that are not answered or do not have endorsed answers.
- We remove the posts that do not contain the HTML tag <code>, because

we cannot extract any API from them.

- We remove the posts that contain code snippets longer than two lines since we focus on single API recommendation in this chapter and code snippets longer than two lines usually contain an API sequence. For multiple code snippets in one post, we remove the post only if all code snippets are longer than two lines.

- We use string matching to find the APIs in the code of each post and remove the posts that do not contain any APIs involved in this chapter, as described in Section 6.3.1.

After the rule-based filtering, we obtained 156,493 Python posts and 148,938 Java posts that contain descriptions of APIs. However, some of the posts are not directly related to API recommendation. For example, some posts only ask about comparing two similar APIs. Unrelated posts are hard to be automatically identified by rules. To ensure the relatedness of the posts in our benchmark dataset, we invite 16 participants with an average of 3 years of development experience in Python or Java for manual checking. For each post, two of the participants are involved in checking the following aspects:

1) whether the query asks about API recommendation;

2) whether the standard APIs recognized by the previous rules are intact, i.e., including the whole class and method names.

3) whether the APIs in answers exactly address the query.

If two participants provide the same answers for one post and also one of the above three aspects is not satisfied, we directly remove the post. If the two participants do not reach an agreement, the post will be forwarded to one of the authors to make a final decision.

As the remaining posts are still too many to be manually checked, we con-

ducted two rounds of annotations. In the first round of annotations, we ask the annotators to label 100 randomly selected posts and conclude the reasons for the cases that they think are not about API recommendation. Then we collect the keywords that frequently appear in these unrelated cases, and remove the posts whose titles contain such keywords. For example, some post titles may contain some specific error names such as "AttributeError: 'Namespace' object has no attribute". We identify these titles and remove them because such posts tend to be related to debugging. However, we will keep the posts if the titles also contain the word "how", since we believe that the posts are likely to ask about error-handling APIs. Although the filtering strategy is coarse, we can remove some noisy posts and facilitate manual annotation. In the second round of annotations, we ask annotators to label all the remaining posts. It takes about one month to complete the two-round annotation process. After both rounds of annotations, we manually checked 13,775 posts, of which 1,262 posts did not reach an agreement with the annotators and needed further checks by one of the authors. We use the commonly-used Fleiss Kappa score [33] to measure the agreement degree between the two annotators and the value is 0.77. The result indicates a high agreement between them. Based on the manual check, 3,245 of the 13,775 labeled posts remain. We take the titles of 3,245 posts as queries following previous studies [16, 76], and finally, we get 1,925 Python queries and 1,320 Java queries. They comprise the first part of our benchmark APIBench-Q, as shown in the second column of Table 6.1.

Mining tutorial websites. API tutorial websites are the second major source of query-API pairs. We choose three popular API tutorial websites GeeksforGeeks [1], Java2s [2], and Kode Java [3] to establish APIBench-Q. Different from Stack Overflow which contains discussion on various topics, API tutorial websites focus on providing examples of how to use APIs. Therefore, manually

Table 6.2: Statistics of Benchmark APIBench-C. The data includes both the training set and the testing set.

PL	Domain	#Projects	#Files	LOC (per func)	#API (per func)	Total number of APIs (only testset)			LOC Threshold of Short Func	LOC Threshold of Long Func
						Standard	User-defined	Popular		
Python	General	899	230,064	15.24	5.55	1,363,240	1,747,878	54,244	8.875	54.875
	ML	323	46,556	13.89	6.08	629,437	339,821	125,377	12.65	46.05
	Security	126	15,785	18.98	6.72	111,393	64,809	3,613	6	86.5
	Web	568	82,771	14.14	5.05	369,114	241,602	11,832	7.35	51.625
	DL	307	39,577	14.58	6.25	413,295	220,228	76,654	11.675	52.525
Java	General	935	1,056,790	11.16	4.06	5,164,481	3,808,124	36,178	6.26	19.2
	Android	377	87,468	8.24	2.91	517,461	267,141	75,069	7.28	16.8
	ML	52	41,377	12.82	4.77	194,013	136,963	0	7.52	19.74
	Testing	55	23,618	9.93	3.98	105,577	55,241	22	6.44	15.68
	Security	58	20,445	12.35	5.32	125,558	74,471	1,243	6.88	20.78

annotating the relatedness of each query to API recommendation is not necessary. We adopt similar rules as mining Stack Overflow to filter out those without code snippets or associated with large code snippets. We finally collect 5,243 Java queries and 2,384 Python queries, which comprise the second part of our benchmark APIBench-Q, as shown in the fifth column of Table 6.1.

Note that we include all queries and corresponding APIs as our test set in APIBench-Q. We do not build a uniform training and validation set for query-based API recommendation approaches because the data sources used by the current work are quite different. For example, using extra data sources is a major contribution for BIKER [76]. Lucene [34] does not need the training set at all. It is hard for us to build a unified training set for training all the approaches. To prevent potential data leakage, we remove the instances that overlap between training sets used by current approaches and APIBench-Q in the preprocessing phase.

Creation of APIBench-C

We create the benchmark dataset APIBench-C by mining GitHub. GitHub [120] is one of the most popular websites for sharing code and includes large numbers of code repositories on different topics and programming languages.

In order to explore the performance of API recommendation under different domains, we first determine the domains for analysis. According to the JetBrains' developer survey and topic labels provided by GitHub[1], we chose four popular domains for Python and Java, respectively, as shown in Table 2. For Python, we consider the domains "Machine Learning" (ML), "Security", "Web", and "Deep Learning" (DL); while for Java, we involve domains "Android", "Machine Learning" (ML), "Testing", and "Security". For each domain, we focus on the repositories tagged with the corresponding topic labels. For example, the "ML" domain only covers the repositories with the "machine learning" tag. As GitHub automatically aggregates all related projects under each domain, we directly collect 500 repositories with the most stars and 500 repositories with the most forks on GitHub[2]. Besides the specific domains, we also built a "General" domain which only considers the popularity of repositories. For the "General" domain, we collect 1,000 repositories with the most stars and 1,000 repositories with the most forks on GitHub regardless of the topics.

Not all the collected repositories are applicable for code-based API recommendation. Some popular repositories do not contain enough code, e.g., only including documentation. To remove such repositories, we use *cloc* [5] to scan the code in each repository and filter out the repositories that 1) have fewer than 10 files or 2) have fewer than 1000 lines of code or 3) have code in Python or Java but with the ratio less than 10%. The number of projects, number of files,

[1]https://github.com/topics

[2]The collection was conducted during April 2021.

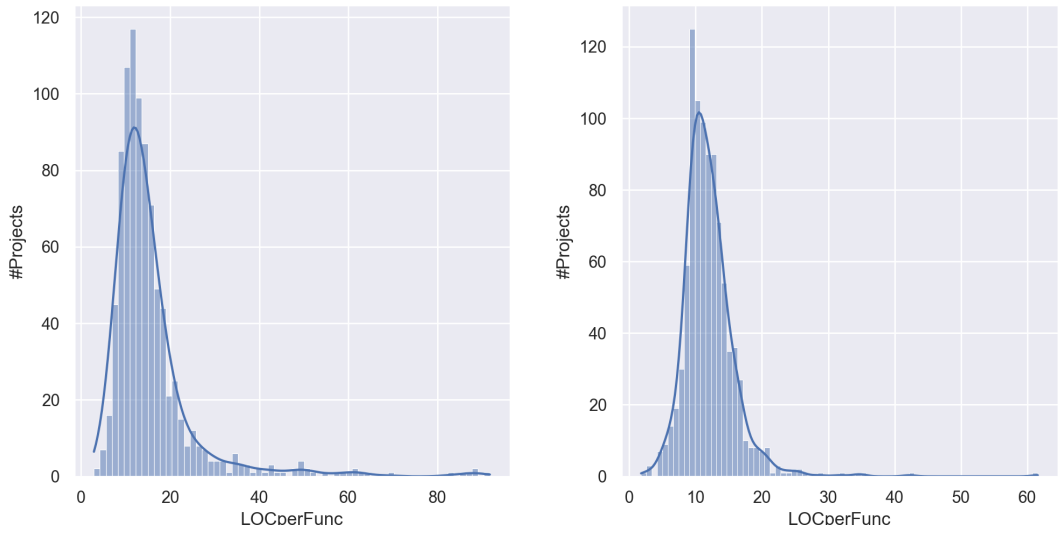


Figure 6.3: Distribution of *code lines per function* for projects under *general* domain (Left: Python, Right: Java).

and average number of code lines for each domain of APIBench-C are shown in Table 6.2.

As most approaches [43, 66, 90, 139, 177] for code-based API recommendation require a training set to learn the API patterns or train the models, we split APIBench-C into a training set and a test set with a ratio of 80% and 20%, respectively. Note that we do not split a project both into the training set and test set, but put all the files of the same project into either the training set or test set, because Alon *et al.* [9] and LeClair *et al.* [97] find that code in the same project usually share the same variable names and code patterns, and splitting without considering project can cause data leakage. For the approaches requiring a validation set, we prepare it from the training set.

In order to study the impact of different recommendation points and different lengths of functions on the performance of current approaches, we analyze the average length of functions in each repository. we leverage Kernel Density

Estimation (KDE) with Gaussian kernels to simulate the distributions. The distributions of the "General" domain for the Python and Java datasets are depicted in Figure 6.3. From the figure, we observe that function lengths are almost normally distributed. Most Python functions contain $5 \sim 30$ lines of code (LOC) and most Java functions contain $5 \sim 20$ lines of code. For studying the impact of function lengths, we divide the functions into extremely short functions, functions of moderate lengths, and extremely long functions according to the confidence interval under 90% confidence level. The confidence interval can be directly calculated by the standard deviations and means. We first determine the confidence interval of functions in different domains using standard deviations and regard the functions with lengths in the confidence interval as functions of moderate lengths. We regard functions with lengths smaller than the confidence interval as extremely short functions and functions with lengths larger than the confidence interval as extremely long functions. Note that except for the study on the impacts of function length, in other experiments we only consider functions of moderate lengths to guarantee that our collected data is representative. The detailed thresholds of confidence intervals for distinguishing extremely long and short functions are illustrated in Table 6.2. We study the impact of function lengths on the performance of code-based approaches in Section 6.5.3.

In order to study the performance of current approaches on different kinds of APIs, we convert the source files of each repository into ASTs and extract all the function calls in them. We label a function call as a *standard* API or *popular* third-party API if it matches one of the APIs collected in Sec. 6.3.1. We label a function call as a *user-defined* API if its implementation can be found in the current repository via import analysis. The average number of API calls per function, number of *standard* APIs, and number of *user-defined* APIs are shown in columns $6 \sim 8$ of Table 6.2, respectively.

6.3.3 Implementation Details

In this section, we describe the details of each approach involved in the benchmark and the metrics for evaluation.

Query reformulation techniques. We choose four popular query reformulation techniques, including Google Prediction Service [59], NLPAUG [113], SEQUER [16], and NLP2API [170]. The detailed description of each technique is illustrated in Table 6.3. Google prediction service is included as one of the most effective approaches in practice, while SEQUER [17] is the state-of-the-art approach. NLPAUG [113] is considered since it is widely used for query reformulation in many NLP studies [87, 134, 168, 237]. We also include NLP2API [171] since it differs from major reformulation methods by first predicting the API class related to the query and then adding the predicted API class into the query.

Table 6.3: The query reformulation techniques and query-based API recommendation approaches involved in the paper. For tools, the years they were last updated are listed. The column name "PL" indicates the applicable programming language.

Approach	Category/ Data Source	PL	Venue	Year
Query Reformulation				
Google Prediction Service [59]	Query expansion, modification	Any	-	2021
NLPAUG [113]	Query expansion, modification	Any	-	2021
SEQUER [16]	Query expansion, modification	Any	ICSE	2021
NLP2API [170]	Query expansion	Java	ICSME	2018
Query-Based API Recommendation				
RACK [172]	Official documentation, Stack Overflow	Java	ICSE	2016
KG-APISumm [109]	Official documentation, Wikipedia	Java	FSE	2019
Naive Baseline	Official documentation	Any	-	2021
DeepAPI [60]	Official documentation	Java	FSE	2016
Lucene [34]	Official documentation	Any	-	2021
BIKER [76]	Official documentation, Stack Overflow	Java	ASE	2018

Query-based API recommendation approaches. We choose five query-based API recommendation approaches published by recent top conferences, in-

cluding KG-APISumm [109], BIKER [76], RACK [172], and DeepAPI [60], along with a popular search library Lucene [34]. The detailed description of each baseline is shown in Table 6.3. We reproduce the five approaches based on the replication packages released by the authors. Besides, we build a naive baseline that recommends APIs by computing the similarities between queries and API descriptions based on BERTOverflow [94]. The native baseline serves as an indicator of the basic performance of similarity-based models. We also notice that different sources are adopted by the approaches for creating the knowledge base. For example, the naive baseline and DeepAPI only consider official documentation, while BIKER and RACK also involve the Q&A forum – Stack Overflow. We list the knowledge source of each approach in Table 6.3. During implementation, we do not align the sources of the approaches, since the sources are claimed as contributions in the original papers. Instead, we design a separate RQ to study the impact of knowledge sources on the performance of API recommendations.

During studying the impact of query reformulation on the recommendation performance, we implement all four query reformulation techniques for each of the six API recommendation baselines because all the baselines do not integrate query reformulation techniques in the original papers.

Table 6.4: The code based API recommendation baselines included in this empirical study. For tools we list the year of its most recent update time. The column name "PL" indicates the applicable programming language.

Approach	Representation	PL	Venue	Year
Practical IDE				
PyCharm [84]	Code tokens	Python	-	2021
Visual Studio Code [121]	Code tokens	Python	-	2021
Eclipse [35]	Code tokens	Java	-	2021
IntelliJ IDEA [83]	Code tokens	Java	-	2021
Approach in Academia				
TravTrans [90]	AST	Python	ICSE	2021
PyART [66]	Token flow Data flow	Python	ICSE	2021
Deep3 [175]	AST, DSL	Python	ICML	2016
FOCUS [139]	API Matrix	Java	ICSE	2019
PAM [42]	API sequence	Java	FSE	2016
PAM-MAX	API sequence	Java	FSE	2016

Code-based API recommendation approaches. We choose four IDEs and five approaches published at recent top conferences as our code-based API recommendation baselines. A detailed description of each baseline is shown in

Table 6.4. For the IDEs and some of the approaches such as TravTrans [90] and Deep3 [175], they can predict any code tokens besides API tokens. In this chapter, we focus on evaluating their performance in recommending APIs. Following prior research [43, 66, 90, 139, 177], we use the training set of APIBench-C to train each of the approaches in academia for a fair comparison.

PAM [43] is the only context-intensive approach, primarily designed for intra-project API pattern mining. In this chapter, we also extend the approach to cross-project recommendation by selecting the best API from projects in the training set for each test case. The extended version of PAM is named as PAM-MAX, which indicates the theoretical maximum performance the context-insensitive approach can achieve.

Evaluation metrics. Since both query-based and code-based API recommendation baselines output a ranked list of candidate APIs, we adopt the commonly used metrics in recommendation tasks for evaluation. The Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Normalized Discounted Cumulative Gain (NDCG) metrics are widely adopted by previous API recommendation studies [76]. In this study, we also involve a new metric Success Rate. The Success Rate@k is defined to evaluate the ability of an approach in recommending correct APIs based on the top-k returned results regardless of the orders. To determine the relevance score in NDCG calculation, we use a relevance score of 1 if an approach hits the correct API class, and a relevance score of 2 if the correct API method is hit. Therefore, we can align the performance of class-level and method-level approaches.

Table 6.5: The basic performance of query-based API recommendation baselines without applying any query reformulation techniques at different metrics (Top-1,3,5,10). Note that we define NDCG as a uniform metric to evaluate class level and method level together, so the NDCG scores listed in two levels have the same values. The red numbers indicate the best performance achieved in top-10 results.

Baseline	**Level**	**Success Rate@k**				**MAP@k**				**MRR**	**NDCG@k**			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
RACK	Class	0.17	0.30	0.35	0.41	0.17	0.23	0.24	0.24	0.25	0.17	0.24	0.26	0.28
KG-APISumm	Class	0.19	0.33	0.40	0.50	0.19	0.25	0.26	0.27	0.28	0.19	0.24	0.27	0.31
Naive Baseline	Class	0.07	0.13	0.16	0.21	0.07	0.10	0.10	0.10	0.11	0.07	0.09	0.10	0.13
	Method	0.02	0.03	0.04	0.05	0.01	0.02	0.03	0.03	0.03	0.07	0.09	0.10	0.13
DeepAPI	Class	0.19	0.27	0.29	0.30	0.19	0.22	0.23	0.23	0.23	0.17	0.22	0.23	0.24
	Method	0.05	0.09	0.10	0.11	0.05	0.07	0.07	0.07	0.07	0.17	0.22	0.23	0.24
Lucene	Class	0.15	0.21	0.24	0.29	0.15	0.17	0.18	0.17	0.19	0.12	0.15	0.16	0.20
	Method	0.04	0.08	0.10	0.14	0.04	0.06	0.06	0.06	0.07	0.12	0.15	0.16	0.20
BIKER	Class	0.33	0.51	0.59	**0.67**	0.33	0.41	0.41	**0.39**	**0.44**	0.27	0.32	0.35	**0.42**
	Method	0.12	0.23	0.29	**0.37**	0.12	0.16	0.18	**0.18**	**0.19**	0.27	0.32	0.35	**0.42**

6.4 Empirical Results of Query Reformulation and Query Based API Recommendation

In this section, we study the RQ 1-3 discussed in Sec. 6.1 and provide the potential findings concluded from the empirical experiments. Since currently no query-based API recommendation approach is specially designed for Python APIs, we focus on studying query-based API recommendation approaches for Java.

6.4.1 Effectiveness of Query-Based API Recommendation Approaches (RQ1-1)

To answer RQ1, we evaluate the six query-based API recommendation baselines listed in Table 6.3 by using the original queries in our benchmark APIBench-Q. The evaluation results are illustrated in Table 6.5.

Class-level v.s. Method-level. Regarding the **class-level** recommendation, as shown in Table 6.5, we can find that BIKER achieves the highest Success Rate, e.g., 0.67 for Success Rate@10, indicating that BIKER is more effective in finding the correct API class in the top-10 returned results for 60%∼70% of cases. Unsurprisingly, the naive baseline shows the worst performance for all the metrics. Even so, the naive baseline can successfully predict the correct API class for around 20% of cases. However, with respect to the **method-level** recommendation, all the approaches show obvious declines. For example, the Success Rate@10 of BIKER is only 0.37, decreasing by 44.8% compared to the class-level recommendation. The Success Rates@10 of DeepAPI and Lucene are only around 0.10, which is far from the requirement of practical development. On average, the approaches fail to give the exact methods for 57.8% APIs that they give the correct classes in top-10 returned recommendations. Thus, recommending method-level APIs still remains a great challenge.

Finding 1: Existing approaches fail to predict 57.8% method-level APIs that could be successfully predicted at the class level. The performance achieved by the approaches is far from the requirement of practical usage. Accurately recommending the method-level APIs still remains a great challenge.

Retrieval-based methods v.s. Learning-based methods. By comparing learning-based methods, such as DeepAPI and naive baseline, with the other retrieval-based methods, we can observe that learning-based methods achieve rel-

atively lower performance regarding the Success Rate@10 metric. For example, on average, retrieval-based methods can accurately predict 46.8% class-level and 25.5% method-level APIs among all the cases in the top-10 returned results, respectively, while learning-based methods can only successfully recommend 25.5% class-level and 8% method-level APIs. A possible reason may be the insufficient training data for the learning-based methods in this task domain. Since there are more than 30,000 APIs from the official documentation, learning-based methods require a large number of query-API pairs for training. However, even the largest Q&A forum, Stack Overflow, contains only about 150,000 posts after our pre-processing, which is not enough for model training.

Finding 2: Learning-based methods do not necessarily outperform retrieval-based methods in recommending more correct APIs. The insufficient query-API pairs for training limit the performance of learning-based methods.

Performance in API ranking. From Table 6.5, we find that there exist obvious gaps between the scores of Success Rate@k and the metrics for evaluating API ranking, such as MAP@k and NDCG@k. For example, RACK achieves a Success Rate@10 score at 0.41, but its MAP@10 score is only 0.24. This indicates that although the approaches are able to find the correct APIs, they cannot well rank them ahead in the returned results. The low MRR scores, e.g., 0.11 $\sim$ 0.44 for class-level API recommendation and 0.03 $\sim$ 0.19 for method-level API recommendation, and NDCG scores also show the poor ranking performance of the approaches. The results manifest that API ranking is still challenging for current approaches.

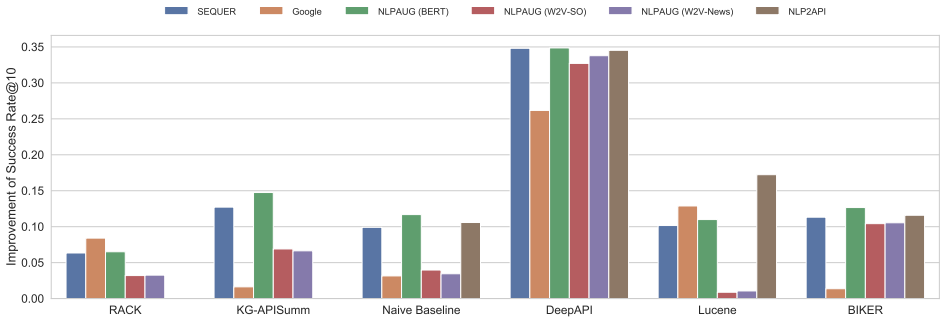
Finding 3: Current approaches cannot rank the correct APIs well, considering the huge gap between the scores of Success Rate and the other ranking metrics.

To sum up, accurately recommending method-level APIs and ranking candidate APIs still remain great challenges. Besides, the insufficient data for training hinders the performance of current learning-based approaches.

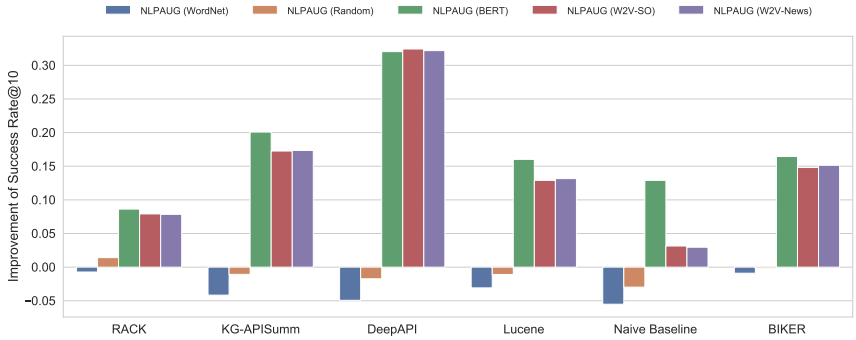
6.4.2 Effectiveness of Query Reformulation Techniques (RQ2)

Original queries can be short in length or contain vague terms. Query reformulation aims at changing original queries for facilitating downstream tasks. In this RQ, we explore the impact of query reformulation on the performance of query-based API recommendation.

We implement the four query reformulation techniques, as listed in Table 6.3, for the original queries. We name the queries reformulated by query expansion techniques and query modification techniques as expanded queries and modified queries, respectively. For each original query, we conduct the reformulation 10 times, producing 10 expanded or modified queries, with the statistics shown in Table 6.1. Note that NLPAUG [113] is a comprehensive data augmentation library for general NLP tasks. We choose the popular word-level insertion and substitution methods designed for manipulating single sentences based on five models, including BERTOverflow [94], Google News Word2vec [58], Stack Overflow Word2vec [205], WordNet [125], and Random model, in the library to generate expanded and modified queries.



(a) Query Expansion



(b) Query Modification

Figure 6.4: The maximum improvement of Success Rate@10 by all query reformulation techniques on **class-level** query-based API recommendation baselines. We do not evaluate the performance of RACK and KG-APISumm in NLP2API reformulated queries as they are only class-level recommendation approaches while NLP2API directly give the predicted API classes. Note that we include Google Prediction Service and SEQUER as expansion techniques here because they expand the queries in most cases.

The queries output by the query reformulation techniques are not ranked in order, and may impact the downstream API recommendation performance variously. To explore the maximum potential effect brought by query reformulation techniques, we evaluate the API recommendation approaches on each reformu-

lated query and choose the best result for analysis. We choose the maximum improvement instead of average improvement for analysis based on the following considerations: 1) It is hard to provide a fair comparison between query reformulation approaches that rank the processed queries such as SEQUER and query reformulation approaches that do not rank the processed queries such as NLPAUG. 2) Query modification would change the query semantics [16, 101]; therefore, using average improvement tends to involve wrong queries and bias the evaluation results. 3) Our goal is to show the potential of current query reformulation approaches, and motivate future research on query reformulation to enhance the performance of API recommendation.

We study the impact of query reformulation on API recommendation from the following two aspects:

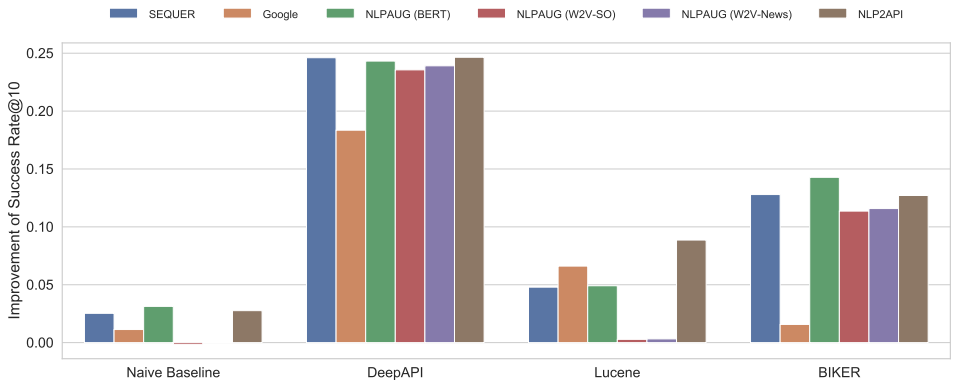
1) whether query reformulation techniques can help predict more correct APIs;

2) whether query reformulation can improve the API ranking performance.

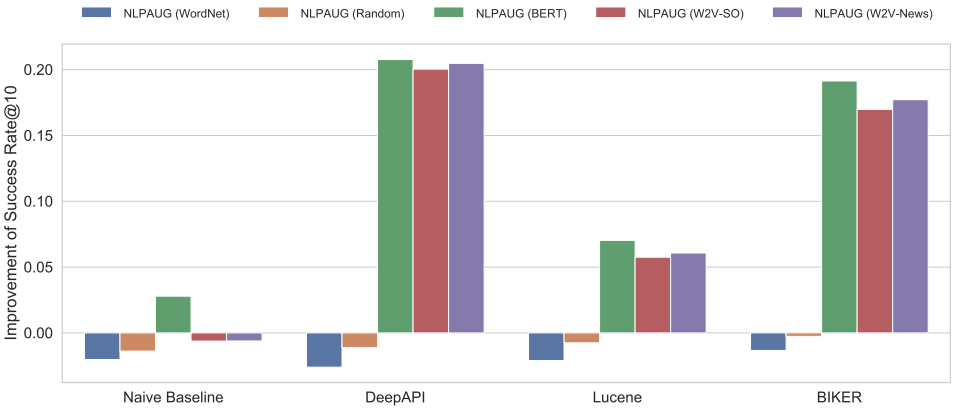
Influence on predicting more correct APIs

With query reformulation v.s. Without query reformulation. The Success Rate metric reflects the proportion of the APIs an approach can correctly predict. The results of implementing the reformulation techniques on API recommendation approaches are illustrated in Figure 6.4 (class-level) and Figure 6.5 (method-level). From the figures, we observe that query reformulation can increase the performance of API recommendation in most cases. Only for a few cases, the performance drops, which can be attributed to the inefficiency of some query reformulation techniques. For example, NLPAUG (WordNet) and NLPAUG (Random) tend to poorly modify the original queries for recommendation, as shown in Figure 6.4 (b) and Figure 6.5 (b). Overall, on average the

process improves the class-level and method-level recommendation by 0.11 and 0.08, which is a corresponding boost of 27.7% and 49.2% compared with the basic performance on original queries.



(a) Query Expansion



(b) Query Modification

Figure 6.5: The maximum improvement of Success Rate@10 by all query reformulation techniques on **method-level** query-based API recommendation baselines.

Finding 4: Query reformulation techniques are quite effective in helping query-based API recommendation approaches give the correct API by adding an average boost of 27.7% and 49.2% on class-level and method-level recommendations,

respectively.

Query expansion v.s. Query modification. By comparing the class-level and method-level recommendation results of query expansion and query modification in Figure 6.4 and Figure 6.5, respectively, we observe that all the query expansion techniques improve the API recommendation performance, but not all the query modification techniques benefit the recommendation. For example, NLPAUG (WordNet) and NLPAUG (Random) generally decrease the performance of current approaches both in class-level and method-level recommendations. This indicates that query expansion techniques bring more stable improvement than query modification techniques. Furthermore, on average, query expansion techniques improve the performance by 0.13 and 0.10 on class-level and method-level recommendation, which is much higher than the improvement of 0.09 and 0.06 achieved by query modification techniques. This also suggests that query expansion techniques are more effective than query modification techniques.

Finding 5: Query expansion is more stable and effective to help current query-based API recommendation approaches give correct APIs than query modification.

Comparing different query expansion techniques. As shown in Figure 6.4 (a) and Figure 6.5 (b), NLP2API and NLPAUG (BERT) present the largest improvement on the performance of query-based API approaches at both class level and method level. For analyzing the improvement, we use two examples to illustrate the query expansion results of NLP2API and NLPAUG (BERT), respectively. In both examples, the most effective approach BIKER fails to predict the API based on the original queries but succeeds given the reformulated queries.

Example 1: Query Expansion	
TECHNIQUE	NLP2API
ORIGINAL QUERY	Returns a new Document instance
PROCESSED QUERY	DocumentBuilderFactory Returns a new Document instance

In the first example, NLP2API expands the query by adding a predicted API class *DocumentBuilderFactory* that is related to the original query. With such an explicit hint, the recommendation approach can narrow down the search scope and pinpoint the requested API method.

Example 2: Query Expansion	
TECHNIQUE	NLPAUG (BERT)
ORIGINAL QUERY	Java reverse string
PROCESSED QUERY	java reverse character string

In the second example, the query is looking for the API *java.lang.StringBuilder.reverse()*, whose description in official documentation is "*Causes this character sequence to be replaced by the reverse of the sequence*". NLPAUG (BERT) adds a relevant word *character* to enrich the semantics of the original query.

Comparing NLPAUG (W2V) with NLPAUG (BERT) and NLP2API in Figure 6.4 and Figure 6.5, we find that the NLPAUG (W2V) is much less effective. To obtain a possible reason for such a difference, we give the third example below. As shown in this example, we find that NLPAUG (W2V) adds two irrelevant words into the original query, which negatively impacts the prediction results of BIKER. This also indicates that contextual embeddings such as BERT are more effective than traditional word embeddings.

Example 3: Query Expansion	
Technique	NLPAUG (W2V-SO)
Original Query	Convert from Radians to Degrees in Java
Processed Query	Convert from AV Radians to Degrees in Long Java

Finding 6: In query expansion, adding predicted API class names or relevant words to queries are more useful than adding other tokens.

Comparing different query modification techniques. Among all query modification techniques, NLPAUG (BERT) presents the biggest improvement on all the baselines at both class level and method level. Example 4 illustrates how NLPAUG (BERT) modifies words in the original query. In the example, the original query asks about ways to calculate the time difference between two dates and the correct API is *java.time.Period.between()*. The description of the API in its official documentation is "*obtains a period consisting of the number of years, months, and days between two dates*". However, the word "*difference*" used in the original query does not clearly describe the functional request. NLPAUG (BERT) modifies the word into "*months*" which exactly appears in the official description. Based on the modifications, the correct API is recommended.

From the second and fourth examples above, we find that BERT-based models show great performance on both query expansion and query modification to help improve the performance of current query-based API recommendation approaches. This indicates that even though the current data source limits the performance of these models to directly predict the correct APIs, they can be used to improve the query quality as query reformulation techniques.

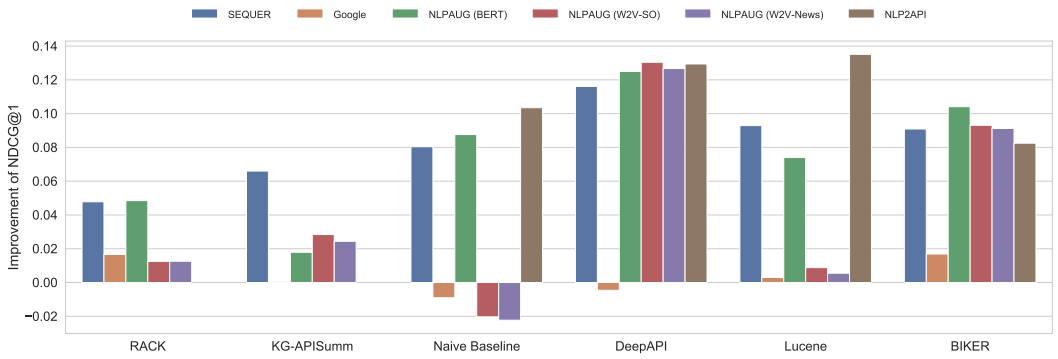
Example 4: Query Modification	
Technique	NLPAUG (BERT)
Original Query	How do I calculate difference between two dates
Processed Query	how do they ~~I~~ calculate months ~~difference~~ between two dates

Finding 7: BERT-based data augmentation shows superior performance in query modification compared with other query modification techniques.

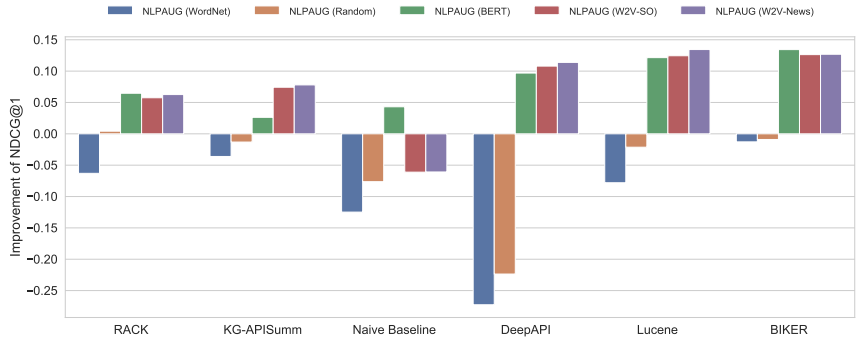
Influence on the performance of API ranking

In this section, we analyze the impact of query reformulation techniques on the performance of API ranking. Since the ideal case is that the correct APIs rank first in the returned results, we use the metric NDCG@1 which considers both class-level and method-level recommendation performance. We compute the changes of NDCG@1 scores for the query-based API recommendation approaches before and after query reformulation. Besides, to focus our analysis on the performance of API ranking instead of the overall recommendation accuracy, the computation is performed only on the cases that are correctly predicted with and without query reformulation.

The results are illustrated in Figure 6.6. As can be seen in Figure 6.6 (a), most query expansion techniques also improve the ranking results of the query-based recommendation approaches. Among all the query expansion techniques, SEQUER, NLPAUG (BERT), RACK and NLP2API can improve the ordering performance relatively better than the others. The biggest improvement of 0.14



(a) Query Expansion



(b) Query Modification

Figure 6.6: The maximum improvement of NDCG@1 by all query reformulation techniques on query-based API recommendation baselines under original successful cases.

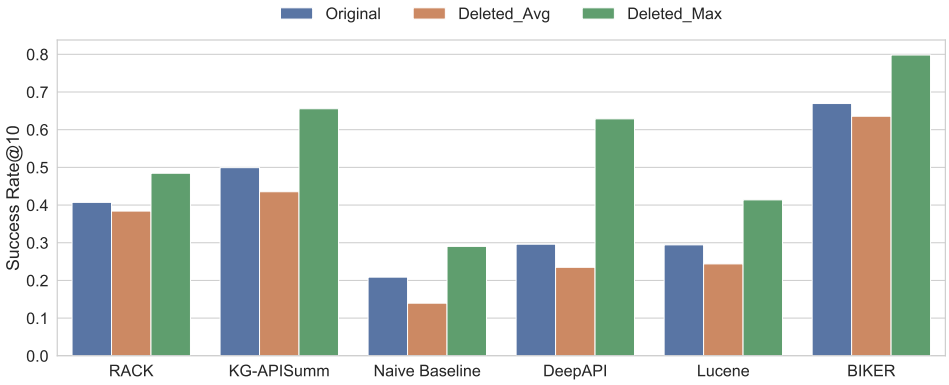
(32% boost) is achieved by NLP2API on the Lucene approach. We also find that on average query expansion also improves MRR by 0.09 (36% boost) and 0.08 (89% boost) on class-level and method-level recommendation, respectively, which indicates that the correct APIs are ranked much higher based on reformulated queries.

According to Figure 6.6 (b), compared with query expansion techniques, query modification techniques are much less effective in improving the API rank-

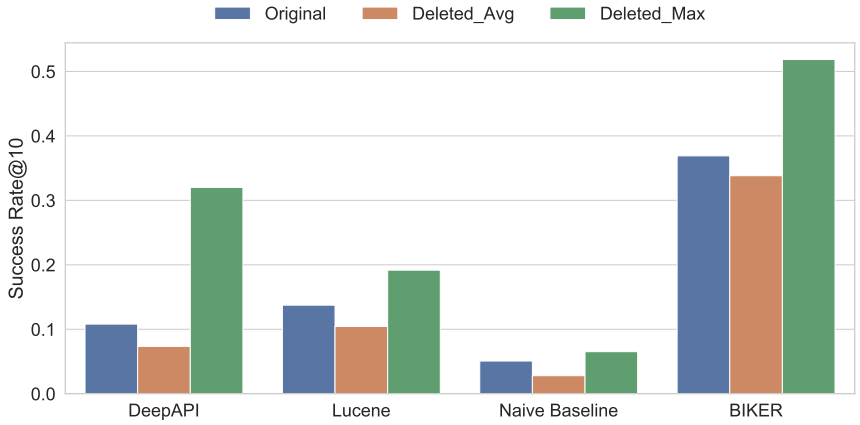
ing performance. For example, the average improvement of NDCG@1 brought by query modification is 0.01 (4% boost), which is 0.06 (14% boost) for query expansion techniques. Comparing different data augmentation methods, we also find that WordNet and random methods tend to negatively impact the ranking results, leading to 24% and 14% drop in terms of NDCG@1, respectively. The results indicate that inappropriate query modification will reduce the ranking performance of the query-based recommendation approaches.

Finding 8: Expanding queries or modifying queries with appropriate data augmentation methods can improve the ranking performance of the query-based API recommendation techniques.

To sum up, query reformulation, especially query expansion, can not only help current approaches recommend more correct APIs, but also improve the ranking performance. However, the reformulation step is generally ignored by current studies. Future work is suggested to involve such a step for more accurate API recommendation.



(a) Class Level



(b) Method Level

Figure 6.7: The maximum and average Success Rate@10 on all baselines when randomly deleting some words in original queries.

A special Query Modification Method: Word Deletion

In previous subsections, we compare and evaluate different query expansion and modification techniques. They aim at enriching the original queries by adding, replacing, or modifying some words without deleting words. In this section, we focus on studying the impact of word deletion, a special query modification method, on the performance of query-based API recommendation ap-

proaches. Different from the previous query reformulation techniques which rely on external data sources, the word deletion method we studied does not leverage any extra knowledge. Our goal is to explore whether original queries contain meaningless or noisy words. Specifically, we randomly delete some words from the original query every time and produce ten different modified queries for one original query.

The maximum and average Success Rate@10 scores based on the modified queries are illustrated in Figure 6.7. As can be seen, the average performance of the query-based API recommendation approaches, denoted as the orange bar, decreases by 0.05 (13% drop) at the class level and 0.03 (18% drop) at the method level. The results are not surprising and indicate that most words in the original queries are helpful for the recommendation. However, the maximum scores, denoted as the green bar, all show that word deletion improves the recommendation performance with an average boost of 38% and 64% for class level and method level, respectively. The improvement demonstrates that the original queries contain noisy words that can bias the recommendation results, although most of the words are useful for recommendation.

After checking all cases, we find that word deletion is helpful for successfully recommending APIs of 545 queries, which maybe attributed to that some noisy words are removed from the original queries. To understand what kinds of words are noisy for the accurate recommendation of these 545 queries, we manually compare the original queries and processed queries. We summarize three possible situations as below:

1) 349 (64%) queries contain unnecessary or meaningless words.

In Example 5, the phrases "*Standard way to*" and "*in java*" are not beneficial for pinpointing the correct API. Stop word removal also has a limited effect on eliminating these words.

Example 5: Word Deletion	
Technique	Random Deletion
Original Query	Standard way to iterate over a StringBuilder in java
Processed Query	~~Standard way to~~ iterate over a StringBuilder ~~in java~~

2) 156 (29%) queries contain too detailed words for explanation.

Example 6: Word Deletion	
Technique	Random Deletion
Original Query	converts a color into a string like 255,0,0
Processed Query	converts a color into a string ~~like 255,0,0~~

In Example 6, the phrase "*like 255,0,0*" is used to explain the "*string*". However, such phrases never appear in the official documentation and the specific number adversely impacts the recommendation results.

3) 34 (6%) queries contain extremely long descriptions.

Based on work [16], most queries have lengths of between one to seven words. In our manual analysis process, one query is regarded as extremely long if it contains more than 10 words. In Example 7, the words after "*while*" actually describe nothing about the task. The long query descriptions can decrease the weight of useful words in the queries thus confusing API recommendation approaches.

Example 7: Word Deletion	
TECHNIQUE	Random Deletion
ORIGINAL QUERY	how to add progress bar to zip utility while zipping or extracting in java
PROCESSED QUERY	how to add progress bar to zip utility ~~while zipping or extracting in java~~

Finding 9: Original queries raised by users usually contain noisy words which can bias the recommendation results, and query reformulation techniques should consider involving noisy-word deletion for a more accurate recommendation.

6.4.3 Data Sources (RQ3)

In RQ1-1, we highlight that insufficient data greatly limits the performance of current learning-based methods. In this section, we conduct a deep analysis on the influence of different data sources on the recommendation results. From Table 6.3, we can observe that current approaches generally leverage three different data sources: official documentation, Q&A forums, and tutorial websites. For analysis, we choose two methods, Lucene and naive baseline, which are flexible to incorporate different data sources. Specifically, we evaluate the methods on the part of queries from the tutorial websites collected in APIBENCH-Q, and the method training is conducted based on the following knowledge base:

1) only official documentation,

2) only Stack Overflow posts, and

3) both official documentation and Stack Overflow posts.

The experiment results are shown in Figure 6.8. As can be seen, training

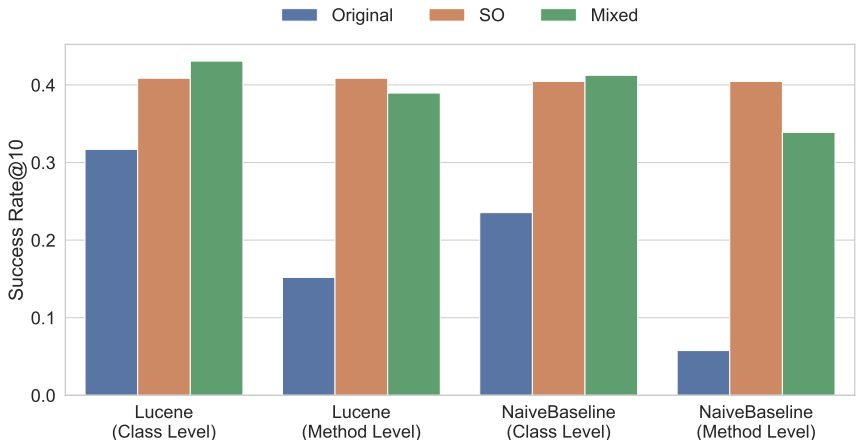


Figure 6.8: The Success Rate@10 of Lucene and Naive Baseline under three data source settings.

on Stack Overflow posts achieves much better performance than on official documentation at both class and method levels. For example, Lucene achieves a 29% boost in class-level and an 169% boost in method-level recommendation when searching based on Stack Overflow than on official documentation; and the naive baseline even achieves a 71% boost in class-level and a 602% boost in method-level recommendation. The advantage of leveraging Stack Overflow posts may be attributed that the discussion on Stack Overflow is more natural and similar to user queries, compared with the descriptions in the official documentation. Besides, the extended usage of some APIs is rarely mentioned in official documentation but is widely discussed in Stack Overflow. An example is used to illustrate the influence of different data sources.

In Example 8, the query asks about the API for generating an MD5 hash of a file. However, there is no standard API specially designed to generate the MD5 hash, so Lucene focuses on two words "*hash*" and "*file*" for recommendation. But the official description of ground truth API *java.security.MessageDigest.digest()* does not contain the word "*file*" since it is a general API that not merely handles

Example 8: Data Source	
Baseline	Lucene
Original Query	Compute the md5 hash of a File
Correct API	java.security.MessageDigest.digest(), java.security.MessageDigest.getInstance()
API Description	Completes the hash computation by performing final operations such as padding
Similar SO Post	How can I generate an MD5 hash in Java?

files. Under this circumstance, Lucene recommends a more relevant but wrong API *java.nio.file.attribute.FileTime.hashCode()*. When involving Stack overflow posts, as there already exists discussion on how to generate the MD5 hash, Lucene can easily pinpoint and recommend the correct API in the posts.

The advantage of leveraging Stack Overflow for recommendation is also demonstrated by the BIKER approach [76], which is the most effective approach in Section 6.4.1. Our finding is consistent with the claim in the work [76] that Stack Overflow posts can mitigate the semantics gap between user queries and official descriptions.

Finding 10: Apart from official documentation, using other data sources such as Stack Overflow can significantly improve the performance of query-based API recommendation approaches.

Table 6.6: The performance of code-based API recommendation baselines at different metrics (Top-1,3,5,10). All baselines are trained and tested on the full dataset from "General" domain of APIBench-C except for PyART. Since PyART takes months to train and test on our full dataset, we randomly sampled 20% of original training and testing testset to evaluate it. The "PL" column indicates the programming language the baselines target. The red number indicates the best performance.

PL	Baseline	Success Rate@k				MAP@k				MRR	NDCG@k			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
Python	TravTrans	0.45	0.57	0.59	**0.62**	0.45	0.50	0.51	**0.51**	**0.51**	0.45	0.52	0.53	**0.54**
	Deep3	0.21	0.34	0.37	0.43	0.20	0.27	0.28	0.28	0.28	0.21	0.29	0.30	0.32
	PyART	0.29	0.38	0.46	0.60	0.29	0.33	0.35	0.37	0.37	0.29	0.34	0.37	0.41
Java	FOCUS	0.01	0.03	0.04	0.06	0.01	0.02	0.02	0.03	0.03	0.01	0.02	0.03	0.04
	PAM	0.01	0.02	0.03	0.05	0.01	0.02	0.02	0.02	0.02	0.01	0.02	0.02	0.03
	PAM-MAX	0.22	0.32	0.36	**0.45**	0.22	0.26	0.27	**0.28**	**0.28**	0.22	0.27	0.29	**0.32**

6.5 Empirical Results of Code-Based API Recommendation

In this section, we study the RQ1 and RQ 4 $\sim$ 6 discussed in Sec 6.1. To study RQ1, RQ4 and RQ5, we evaluate the performance of all the code-based API recommendation approaches on the "General" domain of our benchmark APIBench-C, as shown in Table 6.2, since the "General" domain includes code with different topics and can reflect the overall performance of baselines. For studying the ability of cross-domain adaptation in RQ6, we evaluate the performance of the approaches on all the five domains of our APIBench-C.

6.5.1 Effectiveness of Existing Approaches (RQ1-2)

According to Table 6.4, three approaches for Python and three approaches for Java are evaluated on the "General" domain of APIBench-C. The results are depicted in Table 6.6. We can observe that the learning-based method TravTrans obtains the best performance on the Python dataset, achieving 0.62 and 0.54 for Success Rate@10 and NDCG@10, respectively. The results mean that TravTrans can successfully recommend 62% of APIs in our benchmark and well predict the API rankings. However, the traditional statistical method Deep3 only achieves 0.43 and 0.32 for Success Rate@10 and NDCG@10, respectively, while the pattern-based method FOCUS and PAM achieve less than 0.10 for both Success Rate@10 and NDCG@10. This suggests that learning-based methods obtain superior performance in code-based API recommendation, which is quite different from query-based API recommendation. The possible reason is that lots of well-organized public code repositories provide sufficient data for training code-based API recommendation models.

We also find that FOCUS and PAM show low recommendation accuracy, with all the metric values lower than 0.1. The low performance is attributed to the context representation of the approaches. PAM is a context-insensitive approach, which only mines the top-N APIs that are most likely to be used in the training set and directly recommends them for each file in the test set; while FOCUS takes one step further by extracting the APIs in the test set and building a matrix to match the APIs in the training set. Such coarse-grained context representation or context-insensitive representation does not well capture the relations between APIs. PAM-MAX shows the theoretical best performance context-insensitive methods can achieve. However, the performance of PAM-MAX is still lower than that of TravTrans and PyART which consider fine-grained code features such as code tokens and data flows. The results indicate the effectiveness of fine-grained

Table 6.7: The performance of code-based API recommendation baselines along with 4 widely used IDEs tested on 500 cases sampled from the testset of all domains in APIBench-C. The "PL" column indicates the programming language the baselines target. The red number indicates the best performance. The rows with gray background indicates the performance of IDEs.

PL	Baseline	Success Rate@k				MAP@k				MRR	NDCG@k			
		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
Python	TravTrans	0.38	0.46	0.48	**0.50**	0.38	0.42	0.42	**0.43**	**0.43**	0.38	0.43	0.44	**0.44**
	Deep3	0.19	0.26	0.31	0.38	0.19	0.22	0.23	0.24	0.24	0.19	0.23	0.25	0.28
	PyCharm	0.31	0.42	0.47	0.49	0.31	0.36	0.37	0.37	0.37	0.31	0.38	0.40	0.40
	VSCode	0.05	0.15	0.21	0.35	0.05	0.09	0.11	0.13	0.13	0.05	0.11	0.14	0.18
Java	FOCUS	0.02	0.04	0.05	0.07	0.02	0.03	0.03	0.04	0.04	0.02	0.03	0.04	0.04
	PAM	0.01	0.02	0.05	0.07	0.01	0.02	0.02	0.03	0.03	0.01	0.02	0.03	0.04
	PAM-MAX	0.27	0.38	0.43	0.56	0.27	0.31	0.33	0.34	0.34	0.27	0.33	0.35	0.39
	Eclipse	0.28	0.42	0.49	0.60	0.28	0.34	0.35	0.37	0.37	0.28	0.36	0.39	0.42
	IntelliJ IDEA	0.42	0.58	0.65	**0.67**	0.42	0.49	0.51	**0.51**	**0.51**	0.42	0.51	0.54	**0.55**

approaches for code-based API recommendation.

Besides the recent code-based API recommendation approaches, we also compare the widely-used IDEs. Since it is hard to automatically evaluate IDEs' recommendation performance, we sampled 500 APIs from the original large test set of APIBench-C based on the distribution shown in Table 6.2. We then conduct a manual evaluation by imitating the behaviors of developers on the 500 sampled APIs. We show the results on the sampled test set in Table 6.7. As can be seen, for Python, Pycharm achieves the Success rate@10 at 0.49 and NDCG@10 at 0.40, which is truly competitive to the performance of TravTrans, with Success Rate@10 and NDCG@10 at 0.50 and 0.44, respectively. For Java, IDEs also show competitive performance compared with the baseline approaches. The results demonstrate that the widely-used IDEs are generally effective in API recommendation and far from relying on alphabet orders for recommendation.

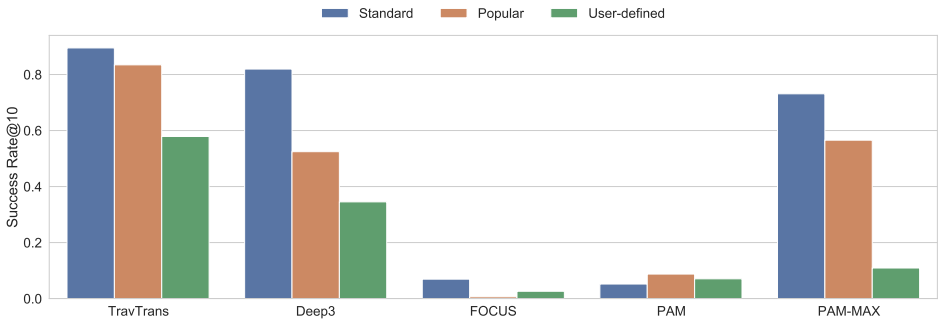


Figure 6.9: The Success Rate@10 of baselines on three categories of APIs at the "General" domain of APIBench-C.

Finding 11: DL models such as TravTrans show superior performance on code-based API recommendation by achieving a Success Rate@10 of 0.62, while widely-used IDEs also obtain satisfying performance by achieving a Success Rate@10 of 0.5 $\sim$ 0.6.

6.5.2 Capability to Recommend Different Kinds of APIs (RQ4)

Exploring which kinds of APIs tend to be wrongly predicted is essential for understanding the bottleneck of current approaches and for providing clues for further improvement. In Section 6.3, we have classified all APIs into standard APIs, popular third-party APIs and user-defined APIs. In this RQ, we study the performance of current baselines for different kinds of APIs. Specifically, we evaluate TravTrans, Deep3, FOCUS, PAM and PAM-MAX on the full test set of the "General" domain, with results shown in Figure 6.9.

As can be seen in Figure 6.9, most approaches achieve a very high Success Rate@10 on standard APIs. For example, TravTrans even successfully recommends more than 90% of standard APIs in the test set. The approaches also present relatively good performance for the popular third-party libraries, e.g.,

TravTrans achieves a Success Rate@10 of more than 0.8. As standard APIs and popular APIs from third-party libraries are widely used in real-world projects, data-driven methods can achieve superior performance. However, it is hard for the approaches to correctly recommend the user-defined APIs as they fail to predict 35.3% $\sim$ 91.3% more of user-defined APIs compared to the prediction of standard APIs.

Finding 12: Although current approaches achieve good performance on recommending standard and popular third-party libraries, they face the challenges of correctly predicting the user-defined APIs.

6.5.3 Capability to Handle Different Contexts (RQ5)

As context representation is an important part of the current code-based API recommendation shown in Figure 6.2, it is worthwhile to study the impact of different contexts on the performance of current approaches. In this RQ, we explore the impact of the following two different types of context.

- lengths of functions, which evaluates the capability of current approaches to handle different lengths of contexts;
- different recommendation points, since different recommendation points affect how much context an approach can be aware of before recommendation.

Capability to handle different lengths of functions. In Section 6.3 and Table 6.2 we classify all functions of APIBench-C into extremely short functions, functions of moderate lengths, or extremely long functions by sampling the first 5%, middle 90%, and last 5% according to the distribution of function lengths. As code-based API recommendation is often based on the context in a function, the length of the function can represent the length of context that

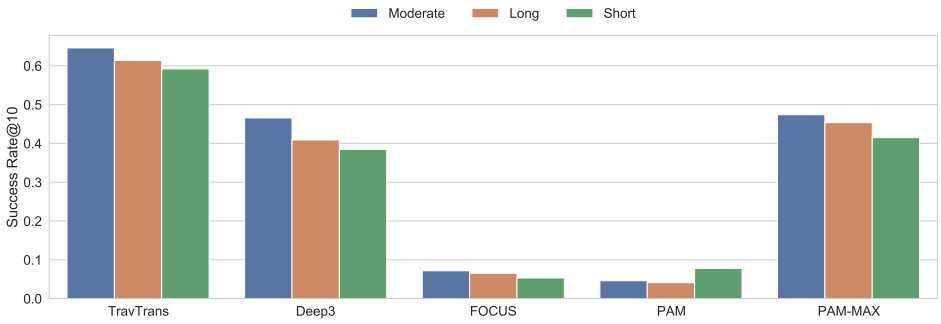


Figure 6.10: The Success Rate@10 of baselines on extremely short, normal, and extremely long contexts at the "General" domain of APIBench-C.

an approach needs to handle. We study the performance of current baselines on functions of different lengths and show the results of TravTrans, Deep3, FOCUS, PAM, and PAM-MAX in Figure 6.10.

From Figure 6.10, we find that most baselines share similar performance distributions on functions of different lengths. They present the best performance on functions with moderate lengths and suffer from performance drops on extremely long or short functions. To be more specific, the performance drops by 7.1% for extremely long functions and 10.6% for extremely short functions on average. The results indicate that context length can affect the performance of current approaches. Besides, the approaches are more difficult to recommend correct APIs for functions of extremely short lengths than those of extremely long lengths.

Finding 13: Context length can impact the performance of current approaches in API recommendation. The approaches perform poorly for the functions with extremely short or long lengths, and accurate recommendation for the extremely short functions is more challenging.

Capability to handle different recommendation points. Similar to the previous work [139], we first define three locations of recommendation points.

Suppose that the LOC of a function is n and the total number of APIs used in the function is m. we define a recommendation point that is on the a_{th} line of the function and is the b_{th} API in the function locates on

1) the front of function if $a/n < 1/4$ and $b/m < 1/4$, or

2) the middle of function if $1/4 < a/n < 3/4$ and $1/4 < b/m < 3/4$, or

3) the back of function if $a/n > 3/4$ and $b/m > 3/4$.

We show an example for illustrating front, middle and back recommendation point in listing 6.1.

```
1  public static void main(String args[]) {
2  //first 1/4 part
3      String[] strArray =
4    new <Front Recommendation Point>
5  ...
6  //middle 1/2 part
7      List l = Arrays.<Middle Recommendation Point>
8  ...
9  //last 1/4 part
10     Collections.<Back Recommendation Point>;
11 ...
12   }
```

Listing 6.1: Example of Recommendation Points

For all the APIs in the test set of the "General" domain, we replace them with placeholders of the above three types of recommendation points for evaluation. We also remove APIs in extremely long or short functions (according to the thresholds shown in Table 6.2) to alleviate the influence of function lengths. We show the results of TravTrans, Deep3, FOCUS, PAM, and PAM-MAX in Figure 6.11.

From Figure 6.11, we observe that current approaches generally perform worse at the front recommendation points by achieving an average Success Rate@10

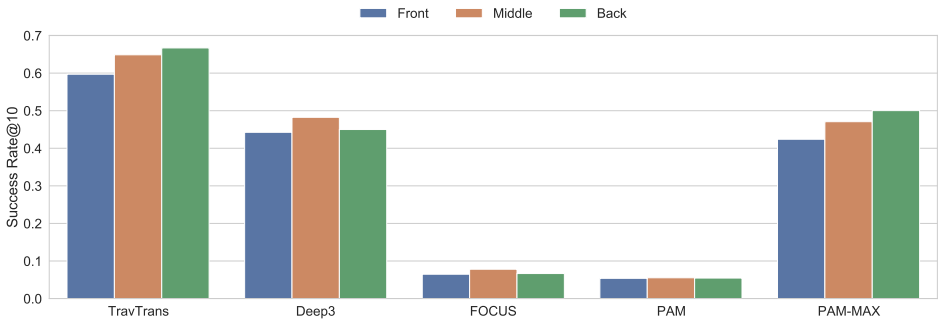


Figure 6.11: The Success Rate@10 of baselines on three categories of recommendation points at the *general* domain of APIBench-C.

of 0.316. This is intuitive since there exists less information for current approaches to leverage at front recommendation points. However, it is worth noting that not all approaches achieve the best performance at the back recommendation point which is associated with the most context among all the recommendation points. The reason may be that the approaches cannot well handle the overwhelming information in long contexts.

Finding 14: The location of recommendation points can affect the performance of current approaches. Current approaches perform worst at front recommendation points due to limited contexts. Some of them also suffer from overwhelming contexts at back recommendation point.

To sum up, different contexts can affect the performance of current code-based API recommendation approaches. Among them, the extremely short contexts and front recommendation points bring the most challenges for the accurate recommendation.

Table 6.8: The cross-domain Success Rate@10 of Python code-based API recommendation baselines. The rows list the domains where three baselines are trained and the columns list the domains where three baselines are evaluated. The red number indicates the best performance an approach achieves when trained on one domain (The largest number in each row). The numbers with gray background indicate the best performance achieved on a specific testing domain (The largest number in each column).

Training Domain	TravTrans				Deep3				PyART			
	ML	Security	Web	DL	ML	Security	Web	DL	ML	Security	Web	DL
ML	0.64	0.58	0.53	**0.71**	0.42	0.41	0.36	**0.48**	0.39	0.35	0.40	**0.40**
Security	0.40	**0.54**	0.54	0.39	0.31	**0.51**	0.42	0.29	0.36	**0.48**	0.47	0.36
Web	0.54	0.63	**0.64**	0.51	0.33	0.42	**0.46**	0.31	0.42	0.47	**0.50**	0.40
DL	0.66	0.58	0.50	**0.68**	0.44	0.39	0.33	**0.44**	0.43	0.36	0.38	**0.45**
General	0.72	0.76	0.78	0.74	0.55	0.65	0.62	0.57	0.44	0.44	0.46	0.46

6.5.4 Adaptation to Cross-Domain Projects (RQ6)

We have divided APIBench-C into five different domains in Section 6.3. In this section, we aim at studying the adaption capability of current approaches for cross-domain projects. We train the approaches in one domain and evaluate them in other different domains. We choose the approaches TravTrans, Deep3, and PyART, which are all designed for Python, for analysis. We do not involve the approaches FOCUS, PAM, or PAM-MAX, since they use coarse-grained context representations or context-insensitive feature, and are difficult to incorporate project-specific information. The first four rows of Table 6.8 list the cross-domain Success Rate@10 of TravTrans, Deep3 and PyART, respectively.

According to Table 6.8, the approaches trained on one domain generally perform best on the test set of the same domain. For example, when trained on data from the "Security" domain, TravTrans, Deep3 and PyART obtain the best

scores at 0.54, 0.51, and 0.48 on the test set of the same domain, respectively, in terms of Success Rate@10. However, their performance drops by 2.1% $\sim$ 43.1% when recommending APIs from different domains.

Finding 15: Current approaches using fine-grained context representation are sensitive to the domain of the training data and suffer from performance drop when recommending cross-domain APIs.

We also analyze the cross-domain performance of the approaches when training on multiple domains instead of on one single domain. Such analysis is worthwhile to explore whether different domains can complement each other. Then we train the approaches on the projects from the "General" domain of APIBench-C and evaluate them on the other four different domains. We show the results in the last row of table 6.8.

From the table, we can see that the approaches trained on the "General" domain generally show the best performance when evaluating on different domains. For example, TravTrans trained on the "General" domain achieves the Success Rate@10 of 0.72, 0.76, 0.78 and 0.74 on ML, Security, Web and DL domains, respectively, which is significantly higher than the corresponding best scores obtained by TravTrans trained on single one domain. We observe an average boost of 14% for the performance of the approaches when trained on multiple domains than on a single domain. The results indicate that training approaches on multiple domains greatly improve the recommendation performance.

Finding 16: Training on multiple domains helps the current approaches to recommend APIs in different single domains, and the performance is generally better than only training on a single domain.

6.6 Implications

6.6.1 Query Reformulation for Query-based API Recommendation

In Sec. 6.4.2, we find that query reformulation techniques can not only help current query-based API recommendation approaches find more correct APIs but also improve the ranking performance. Based on query reformulation, BIKER can even achieve a Success Rate@10 of 0.80 at class-level and 0.51 in method-level API recommendation. The results demonstrate that query quality has a great impact on the recommendation results and suggest that query reformulation should become a common pre-processing technique used before query-based API recommendation. We also discover that some query reformulation techniques, such as adding predicted API class names or relevant words, can improve the performance of query-based API recommendation approaches. However, to the best of our knowledge, few studies have considered integrating these techniques, which could be one major reason that current approaches achieve limited performance.

By implementing a random deletion strategy, in Sec. 6.4.2 we find that user queries usually contain noisy words, which can bias the recommendation results. We summarize three kinds of cases in which a query contains noisy words. However, there exists very little work that aims to detect and eliminate irrelevant words for recommendation systems, which poses a great challenge for current approaches to be robust when handling various user queries. Although a random deletion strategy reduces the overall performance on average, the positive improvement of deletion on some specific words indicates the potential benefits of noisy word deletion.

Implication 1: *Current query-based API recommendation approaches should be integrated with query reformulation techniques to be more effective.*

6.6.2 Data Sources for Query-based API Recommendation

In Sec. 6.4.1, we point out that current query-based API recommendation approaches face the problem of building a comprehensive knowledge base due to the lack of enough data such as query-API pairs. In Sec. 6.4.3, we further discover that there is a semantic gap between user queries and descriptions from the official documentation. Both the lack of enough data for knowledge base creation and the semantic gap increase the difficulty of accurate API recommendation based on only official documentation. Such challenges can not be easily solved by improving learning-based models or pattern-based models. One effective way to mitigate the difficulty is to involve Stack Overflow posts, as analyzed in Section 6.4.3. While Stack Overflow is only one type of data source, our analysis demonstrates that adding appropriate data sources can improve the performance of query-based API recommendation approaches.

Implication 2: *Apart from query reformulation, adding appropriate data sources provides another solution to bridge the gap between queries and APIs.*

6.6.3 Low Resource Setting in Query-based API Recommendation

In Section 6.4.1, we find that current learning-based methods do not necessarily outperform traditional retrieval-based methods. We attribute the results to

the limited data such as query-API pairs in the query-based API recommendation task, which is a low-resource scenario [24, 67]. We also discover that pre-trained models such as BERT show superior performance in query reformulation in Section 6.4.2. This indicates that current pre-trained models can implicitly mitigate the semantic gap between user queries and official descriptions of APIs. Future work is suggested to explore how to make the best use of pre-trained models for query-based API recommendation based on limited available data.

Implication 3: *Few-shot learning with powerful pre-trained models can be a solution to further improve the performance of query-based API recommendation.*

6.6.4 User-defined APIs

In Section 6.5.2, we find that current code-based API recommendation approaches, no matter pattern-based or learning-based models, all face the challenge of recommending user-defined APIs. User-defined APIs have become the major bottleneck to further improve the performance of current code-based API recommendation approaches. However, as user-defined APIs usually do not appear in the training set, they can hardly be learned by machine learning methods or be mined by pattern-based methods. A possible solution used by current approaches [76, 91] is to regard the API as a code token and predict the token based on previous contexts. However, this solution also fails if the API token never appears in previous context. Thus, accurately predicting user-defined APIs should be one major direction of code-based API recommendation in future work.

Implication 4: *User-defined API recommendation is one major bottleneck for improving the performance of current code-based API recommendation approaches and remains unsolved.*

6.6.5 Query-based API Recommendation with Usage Patterns

In this chapter, we only focus on testing whether an approach can recommend the correct APIs, but we believe developers can always benefit more from detailed information about how to use the recommended APIs. A common method is to provide summaries such as the signature and constraints extracted from official documentation along with the recommended APIs. For example, KG-APISumm proposed by Liu *et al.* [109] provides a detailed summary of the recommended API class. However, official documentation sometimes cannot provide enough usage information about an API, which may cause API misuse. For instance, a fresh developer may search "*how to read a file*" in Python and the recommended API should be *fileObject.read()*, but without sufficient experience to use file operations, the developers may forget to close the file after reading it.

A possible solution to complement official documentation and avoid possible API misuse is to provide usage patterns from other developers. In the above example, a common usage pattern *open()*, *fileObject.read()*, *fileObject.close()* can prevent dangerous file operations. As there exist some pattern mining approaches on code, we can combine query-based API recommendation with code-based pattern mining methods for better providing the usage pattern.

Implication 5: *Code-based API recommendation approaches can provide usage patterns to enrich the results returned by query-based API recommendation approaches.*

6.6.6 Implications for Different Group of Software Practitioners

In this subsection, we conclude some implications for different group of software practitioners.

Software Researchers. For query-based API recommendation, we conclude that query reformulation techniques can bring significant improvement for current API recommendation approaches in Section 6.4.2. Despite of the effectiveness of query reformulation, it still remains unexplored on the factors that impact the performance of the technique. We believe a comprehensive study of query reformulation can be an important future direction for API recommendation. For code-based API recommendation, we find that the major bottleneck for current approaches is user-defined API recommendation in Section 6.5.2. We suggest software researchers to focus more on the user-defined API recommendation for improving the practicability of API recommendation approaches.

Software Developers. As illustrated in Section 6.4.3, there exists a knowledge gap between official documentation and user queries, which limits the performance of current query-based API recommendation approaches. For developers who design new APIs, we believe adding more practical examples in the documentation or using more natural language descriptions would mitigate the knowledge gap. In Section 6.4.2, we find that current queries sometimes contain unnecessary information that confuses the API recommendation approaches. For developers

who search for APIs, we believe that creating a query by using more professional words instead of unnecessary long descriptions can facilitate the search process.

6.7 Threat To Validity

In this section, we describe the possible threats we may face in this study and discuss how we mitigate them.

6.7.1 Internal Validity

Our research may face the following internal threats:

Baseline Re-implementation. In this chapter, we re-implemented several baselines according to the code or replication packages released by their authors. However, as some baselines are not primarily designed for API recommendation, we slightly modified their code and adapted them into our task and our benchmark. For example, we limit the prediction scope of code completion baselines to only API tokens. Such adaptations may cause the performance of baselines to be slightly different from those in the original papers. To mitigate this threat and validate the correctness of our re-implementation, we refer to some related work that cites these baselines and confirm our experiment results with them.

Data Quality. We build APIBench-Q by manually selecting and labeling API-related queries from Stack Overflow and some tutorial websites. This process involved some human checks so that some subjective factors may influence the quality of our datasets. To mitigate this threat, we involve at least two persons to label one case and let one of our authors further check if the previous two persons give different opinions to the case. We also implement some rules to automatically filter out the cases that are explicitly unrelated to API recommendation.

Identification of User-defined APIs. We utilize commonly used static

import analysis to analyze the import statements in each source file and try to identify the implementation of imported libraries. To guarantee the quality of our benchmark dataset, we regard an API as a user-defined API only if we can find its implementation. However, since the completeness of static import analysis is still an open challenge, there may exist several user-defined APIs that cannot be identified. We will further refine the dataset when more advanced static important analysis tools are available.

6.7.2 External Validity

Our research may face the following external threats:

Data Selection. To the best of our knowledge, APIBench is the largest benchmark in API recommendation task. We try to make it more representative by selecting real-world code repositories from the most popular domains at GitHub and real-world queries from the largest Q&A forum StackOverflow according to several developer surveys [82, 192]. All findings in this empirical study are based on this dataset. However, there may still be slight differences when adapting our findings into other domains and datasets that we do not discuss in this chapter.

Programming Language. Our study focuses on the API recommendation for Python and Java, and the findings included in this study may not be generalized to other programming languages. However, we believe the impacts of programming languages should not be significant as Python and Java are the most representative dynamically typed and statically typed languages, respectively.

6.8 Conclusion

In this chapter, we present an empirical study on the API recommendation task. We classify current work into query-based and code-based API recommendation and build a benchmark named VLibs to align the performance of different recommendation approaches. We conclude some findings based on the empirical results of current approaches.

For query-based API recommendation approaches, we find that 1) recommending method-level APIs is still challenging; 2) query reformulation techniques have great potential to improve the quality of user queries thus they can help current approaches better recommend APIs. What's more, user queries also contain some meaningless and verbose words and even a simple word deletion method can improve the performance; 3) approaches built upon different data sources have quite different performances. Q&A forums such as Stack Overflow can greatly help mitigate the gap between user queries and API descriptions.

For code-based API recommendation, we emphasize the superior performance of current deep learning models such as Transformer. However, they still face the challenge of recommending user-defined APIs. We also find different contexts, such as different locations of recommendation points and context length, can impact the performance of current approaches. Besides, current approaches suffer from recommending cross-domain APIs.

Based on the findings, we summarize some future directions for improving the performance of API recommendation. For query-based approaches, we encourage researchers to integrate query reformulation techniques with query-based API recommendation approaches to obtain better performance, but how to choose the best query reformulation strategy still remains as future work. We also believe some few-shot learning methods and different data sources can bridge the gap between user queries and knowledge base under low-resource scenarios. For

code-based approaches, we recommend future work focusing on improving the performance of user-defined API recommendation and training the approach on multiple domains instead of a single domain.

Apart from the findings and implications concluded in this chapter, we also identify some future work that can be conducted for API recommendation. First, our paper focuses on benchmarking and provides an objective evaluation for all approaches. However, some approaches provide summaries for the recommended APIs that are not assessed in our study. For such approaches, subjective evaluation such as a developer survey can be conducted to verify the quality of recommended API descriptions and usage information. Future work could consider complementing our work. Second, our empirical results show that query reformulation techniques are quite effective in improving the query quality. As this chapter mainly focuses on API recommendation, we do not discuss different query reformulation techniques comprehensively. Future work can focus on studying the query reformulation techniques for facilitating downstream tasks.

Chapter 7

Source-level Python Run-time Environment Conflict Detection

It is beneficial for developers to check the correctness of their configuration files before publishing them to the users. Current version-level run-time environment conflict detection approaches can only check the errors in the configuration files but ignore the incompatibility between the configuration file and the software. In this chapter, we focus on checking the potential run-time environment conflicts caused due to the incompatibility between the built run-time environments and the software. The main points of this chapter are as follows. (1) We study the source-level configuration issues in the PyPI ecosystem systematically. (2) We propose an automatic approach PyConf that incorporates *Installation Check*, *Dependency Check* and *Import Validation* to detect configuration issues for Python projects. (3) We build a benchmark VLibs that includes 131,720 library releases to facilitate the evaluation of automatic dependency inference approaches.

7.1 Introduction

Python has experienced a remarkable 22.5% year-over-year surge in usage, positioning it as the second most favored programming language within the GitHub open-source community [56]. The popularity of Python is primarily established by its flexible and readable syntax, making it easier for developers to maintain complicated software. Nowadays, the success of Python owes much to its thriving and supportive community, which plays a pivotal role in fostering its prosperity. The accessibility and utility of Python are further amplified by the public libraries available on the Python Package Index (PyPI) platform. With over 470 thousand Python projects and more than 4.7 million releases [41], PyPI serves as the primary repository for numerous third-party libraries. By encapsulating reusable functionalities with APIs in third-party libraries, developers can easily build complicated applications.

In the dynamic ecosystem of third-party libraries hosted on PyPI, multiple releases of the same library are often available, distinguished by version numbers. To use a specific library release, developers must specify both the library's name and the desired version. Utilizing the official library management tool, `pip` [39], for PyPI, developers can effortlessly retrieve and install the intended release based on the associated configurations. Once installed in the current run-time environment, the library release can be accessed through import statements within the source code. Compared with static programming languages such as Java and C/C++, third-party library usage in Python is much simpler and requires no compilation. However, even with this streamlined approach, the presence of any configuration issues in the third-party libraries can lead to potential run-time failures.

Numerous research efforts [11, 150, 193, 213, 214, 215] are dedicated to detecting potential dependency conflicts among diverse third-party libraries during

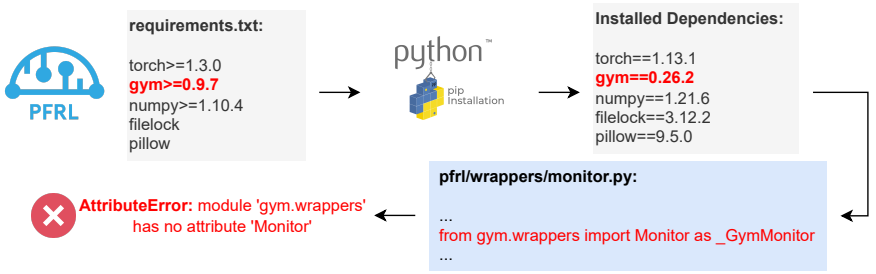


Figure 7.1: A configuration issue of the third-party library PFRL.

the constraint-solving process. As a library may rely on others, a dependency graph can be established to represent the interconnected libraries with nodes and version constraints with edges. Based on the dependency graph, these approaches use SMT solvers to determine an available version assignment for each library. In addition, some other work [22, 71, 241] develops knowledge graphs for third-party libraries on PyPI and then builds run-time environments for new Python projects based on the knowledge graphs.

The aforementioned version-level approaches have been established and evaluated under the assumption that the configurations of existing Python projects are accurate, as they solely examine version constraints in configurations without inspecting the source code. However, we have discovered instances where this assumption does not hold, and we present an illustrative example in Fig. 7.1. In this example, the third-party library `PFRL` [45] implements several well-known reinforcement learning algorithms. It records all required third-party libraries in the `requirements.txt` file. During installation, the library manager `pip` resolves the constraints in `requirements.txt` and installs the latest available version for each library. A widely-used library, `gym`, is among the dependencies specified in `requirements.txt`. The configuration for `gym` merely requires a version newer than 0.9.7. As a result, `pip` installs the latest version, 0.26.2, into the project as

it satisfies the constraint[1]. Since version 0.26.2 of `gym` does not conflict with other libraries in `requirements.txt`, it passes the regular conflict check and becomes part of the run-time environment. However, when running the code in `PFRL`, an `AttributeError` is raised as the `Monitor` class from `gym.wrappers` in the file `pfrl/wrappers/monitor.py` cannot be found. This issue is widely discussed on `PFRL`'s GitHub issues [198] and Stack Overflow [96]. The root cause is that `gym` removed the `Monitor` class starting from version 0.23.0. Since this change only affects the source code and is not detected by version-level checks, the problem remains unnoticed. This scenario highlights the inadequacy of version-level checks in ensuring the compatibility of source code and run-time environments. To address this problem, the library `gym` should be constrained to versions `gym>=0.9.7, gym<0.23.0`. However, predicting such changes in `gym` during the development of `PFRL` is not feasible since version 0.23.0 of `gym` had not been released at that time. Therefore, the configurations in Python projects can be outdated despite being correct at the release time.

The above-mentioned challenge of version-level dependency checks may pose big threats to the development and evaluation of automatic dependency inference approaches that heavily depend on PyPI library configurations. To address this challenge, we first comprehensively study the potential configuration issues in the PyPI ecosystem (RQ1) and then construct a source-level compatible dataset to facilitate the evaluation of existing automatic dependency inference approaches (RQ2).

To answer RQ1, we introduce PyConf, an automatic approach designed to identify both version-level and source-level configuration issues in third-party libraries on the PyPI platform. PyConf incorporates three distinct checks, namely *Installation Check*, *Dependency Check* and *Import Validation*, to detect

[1]The installation was performed in July 2023.

configuration issues during the setup stage, packing stage and usage stage of third-party libraries, respectively. Through an analysis of PyConf's results, we identify 183,864 (54%) library releases among the 338,069 checked releases that exhibit potential configuration issues. Notably, 68% of these issues are newly detected by the source-level check, i.e., the *Import Validation*. We identify 15 kinds of configuration issues based on the run-time error types and classify them into three major categories: *Incomplete Configuration*, *Incorrect Configuration* and *Incorrect Code*. For RQ2, we construct a benchmark, VLibs, consisting of 131,720 library releases that successfully pass all three checks implemented by PyConf. We then evaluate the correctness of the inferred run-time environments by the three state-of-the-art automatic dependency inference approaches Pipreqs [156], Dockerizeme [71] and PyEGo [241], respectively.

Key Findings. Based on a thorough analysis of the experiment results pertaining to RQ1 and RQ2, we have summarized the following key findings:

1) Developers tend to provide inadequate configurations for the usage of libraries, especially for Python versions and direct imports in source code.

2) Developers make mistakes in writing configurations since 19% of configuration issues are incorrect configurations. What's more, about 50% incorrect configuration issues can only be detected by *Import Validation*, indicating the importance of source-level validation.

3) Current automatic dependency inference approaches fail to infer about 35% of Python projects. Among the failures, the majority are attributed to dependency conflicts and the absence of required libraries in the generated configurations.

Based on the findings, we conclude two implications for the developers of third-party libraries on the PyPI platform and the future research of automatic dependency inference. Specifically, we find that "*less is more*", i.e., fewer depen-

dency constraints can lead to more configuration errors, so we suggest developers avoid employing open constraints such as `version>1.0`, but set complete and strict dependency constraints limiting the versions of dependencies to the verified ones before the release dates. For future research on automatic dependency inference, we suggest researchers add more conflict checks to avoid generating incorrect configurations.

Contributions. To sum up, we list our contributions as follows.

- To the best of our knowledge, we are the first to study the source-level configuration issues in the PyPI ecosystem systematically.
- We propose an automatic approach PyConf that incorporates *Installation Check*, *Dependency Check* and *Import Validation* to detect configuration issues for Python projects.
- We build a benchmark VLibs that includes 131,720 library releases to facilitate the evaluation of automatic dependency inference approaches.

7.2 Methodology

In this section, we introduce how we collect the metadata of PyPI libraries and how PyConf works to detect potential configuration issues.

7.2.1 Data Preparation

As of July 2023, the PyPI ecosystem boasts a substantial collection of approximately 471,000 libraries, encompassing over 4,712,000 releases [41]. It is quite difficult to perform a comprehensive analysis of all the libraries and their releases on a single machine. To address this challenge, we employ a well-established strategy used in prior studies [22, 71, 241] and collect data from the top 10,000

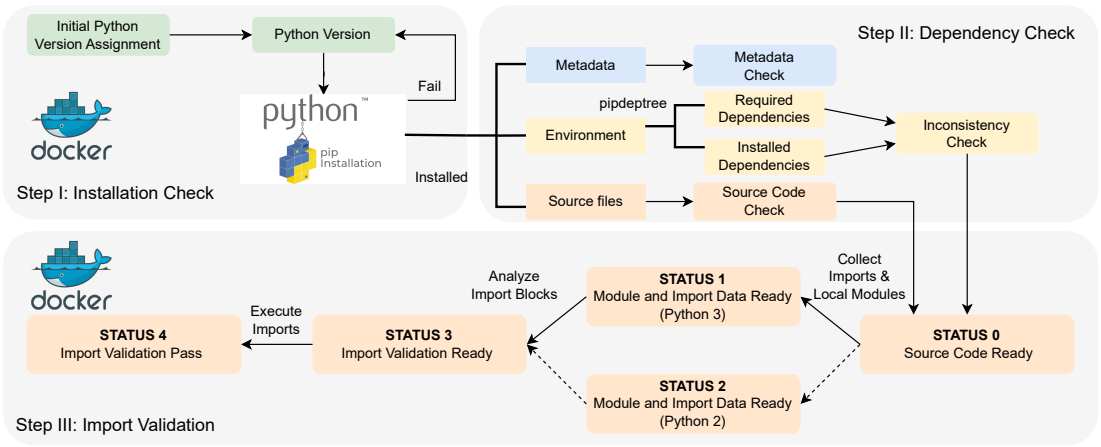


Figure 7.2: The overview of PyConf.

most popular libraries, as reported by libraries.io [200]. Libraries with only one or two releases, which generally do not necessitate automatic version determination, are excluded from our analysis, resulting in a dataset comprising 8,282 libraries and 338,069 releases[2]. The first column of Table 7.1 provides statistics on these libraries. In accordance with Python Enhancement Proposal (PEP) 508 [23], names of PyPI libraries are case-insensitive, and distinctions between dash, dot, and underscore are disregarded. To ensure consistency and avoid multiple names for the same library, we normalize all library names to lowercase and replace all dots and underscores with dashes.

Initial Python Version Assignment. As mentioned in Chp. 2, a smooth `pip` installation requires the correct Python version. Hence, we begin by assigning an initial Python version to each library release in the dataset. To acquire the Python version constraints for each library release, we examine the classifiers set by developers on the project web page of the PyPI platform. PyPI offers a set of classifiers for developers to denote the compatibility status of library releases. Among these classifiers, those categorized under the programming

[2]Data was collected in November 2022.

Table 7.1: The statistics of the PyPI libraries in our study. "Installed" and "Validated" indicate the libraries passing the Dependency Check and all checks of PyConf, respectively. #Stars indicate the number of GitHub Stars of libraries. #Stars, #Classes, #Functions and #Imports are shown in the format of Avg/-Max/Min. The data of #Stars is calculated per library and others are calculated per release. Note that the source code data in the first column is not available as the libraries are not installed.

	All	**Installed**	**Validated** (VLibs)
#Libraries	8,282	7,830	5,371
#Releases	338,069	303,377	131,720
#Modules	-	368,304	144,250
#Stars (k)	2.3/159.0/0.0	-	-
#Classes (k)	-	0.3/88.1/0.0	0.2/21.9/0.0
#Functions (k)	-	1.5/261.4/0.0	0.6/50.3/0.0
#External Imports	-	49/2207/0	23/386/0
#Lines of Code (k)	-	18.2/7455.3/0.0	6.5/551.3/0.0

language category specify the Python versions with which a library release is compatible. For instance, the developers of library release `pipreqs-0.4.13` add classifier `Python::3.7` in the web page [144], indicating that `pipreqs-0.4.13` can be used in Python version 3.7. By collecting such classifiers from the web pages, we determine the latest Python version applicable to each library release as the initial Python version.

The initial Python versions inferred from classifiers provide relatively reliable insights into developers' intentions regarding library usage. However, setting classifiers is not mandatory when developers publish a new release on PyPI, so we cannot assign initial Python versions for certain library releases lacking

appropriate classifiers. To tackle this problem, we collect the release dates of such library releases and select the latest Python version released 180 days before the release dates of the library releases. This assignment may not be accurate but can be fixed by PyConf when the installation fails.

7.2.2 PyConf: Detecting Configuration Issues

PyConf checks both version-level and source-level configuration issues for libraries in the PyPI ecosystem. We present the overview of PyConf in Fig. 7.2. PyConf conducts three checks, namely *Installation Check*, *Dependency Check* and *Import Validation*, to discover potential configuration issues in the setup stage, the packing stage and the usage stage of libraries, respectively. The *Installation Check* verifies the availability of the library releases and detects fatal configuration errors, such as dependency conflicts, that even prevent successful library installation. The *Dependency Check* verifies the consistency of the installed environment with the specified configuration, correctness of the library metadata and syntactic correctness of the source code, which are threatened by mistakes made during the packing stage before a library is published. The *Import Validation* verifies the compatibility of the source code with the installed run-time environment to discover run-time errors during the usage of libraries.

Installation Check. Upon receiving the name and version of a library, PyConf initiates an empty run-time environment within a docker container using the initial Python version. The library release is then installed using the command `pip install <library> == <version>`. However, due to certain initial Python version assignments being estimated based on release dates, and the existence of erroneous configurations authored by developers, some library releases may fail to be installed under the initial Python version. For libraries with Python version constraints, PyConf retries the installation using another valid

Python version. For libraries lacking such constraints or failing on all versions indicated by the constraints, PyConf adopts a heuristic searching approach to minimize overhead. Specifically, PyConf first copies the Python version of other successfully installed releases of the same library as different releases of the same library require similar run-time environments. In cases where the installation still fails, PyConf attempts commonly used versions such as 2.7, 3.6, and 3.10. The heuristic searching strategy can handle most installation failures and PyConf resorts to trying all possible Python versions only when the heuristic search proves unsuccessful. Therefore, the installation check fails only when there is no compatible Python version for the given library release or when there are critical errors in applying the configurations provided by developers. This indicates that the library release is not available for use under any Python version.

Dependency Check. During the installation process of a library release via `pip`, three types of data are downloaded into the system:

1) *Metadata.* The metadata is stored in the format of a folder named `<package> -<version>.dist-info`. To analyze this metadata, PyConf focuses on the `top_level.txt` file, which enumerates all modules that can be imported from the library release.

2) *Run-time environment.* PyConf captures information regarding the installed run-time environment, including the versions of installed third-party libraries and the version constraints of the required third-party libraries, via the `pipdeptree` [202] tool. PyConf then proceeds to resolve the version constraints of the required third-party libraries and cross-checks them against the installed versions to detect potential inconsistencies.

3) *Source files.* To validate the syntactic correctness of the source code, PyConf locates source folders or files based on the modules collected from the metadata. It employs the `ast` module [40] to parse all source files and identifies

the presence of any syntax errors.

Algorithm 4 Import Block Analysis

Input: Abstract Syntax Tree (AST) of the current source file, *ast*;

Output: Import blocks, B; Block-free Imports, D;

```
1: function GETIMPORTBLOCKS(block)                    ▷ The main function
2:     importBlocks ← {}; subBlocks ← divideBlock(block)
3:     for sb ∈ subBlocks do
4:         curIB ← {}; curBFI ← {}
5:         for node ∈ sb.importnodes do
6:             if isIforTryOutside(node) then
7:                 bNode ← getOutmostIforTryNode(node)
8:                 curIB ← curIB + {GETIMPORTBLOCKS(bNode)}
9:             else
10:                curBFI ← curBFI + {node}
11:            end if
12:        end for
13:        curB ← curIB + {curBFI}
14:        importBlocks ← importBlocks + {curB}
15:    end for
16:    return importBlocks
17: end function
18: blocks ← { }; D ← { }; B ← { }                     ▷ The overall algorithm
19: for node ∈ ast.importnodes do
20:    if isIforTryOutside(node) then
21:        blocks ← blocks + {getOutmostIforTryNode(node)}
22:    else
23:        D ← D + {node}
```

24: **end if**

25: **end for**

26: **for** $b \in blocks$ **do**

27: $B \leftarrow B + \{\text{GETIMPORTBLOCKS}(b)\}$

28: **end for**

Import Validation. Successful installation and consistent run-time environment do not necessarily guarantee the smooth usage of the library, since the execution still fails if some external import requirements in the source code cannot be fulfilled. PyConf conducts *Import Validation* to detect these issues. PyConf leverages a finite state machine (FSM) with four states to guide the process of *Import Validation*, as shown in step III of Fig. 7.2.

1) Collect Imports and Local Modules (STATUS 0 → STATUS 1/2). Initially, all library releases enter STATUS 0 if PyConf can successfully locate their source code in *Dependency Check*. For library releases with STATUS 0, PyConf collects import statements in the source code. Import statements in the source code can be of two types: *internal imports*, which introduce local modules within the project, and *external imports*, which require third-party libraries from the run-time environment. PyConf employs different approaches to handle the two kinds of import statements.

Local modules are required to distinguish internal imports and external imports. Different source files also have different available local modules. For each source file in the library release, PyConf collects the names of all Python source files and the sub-directories with `__init__.py` file in the same directory, as well as image files such as `.so` and `.pyd`, as local modules. Next, PyConf checks all import statements in the source file and compares the imported module with the local modules to identify internal imports. Since internal imports are not pertinent to the run-time environment, they are excluded from the *Import Validation*

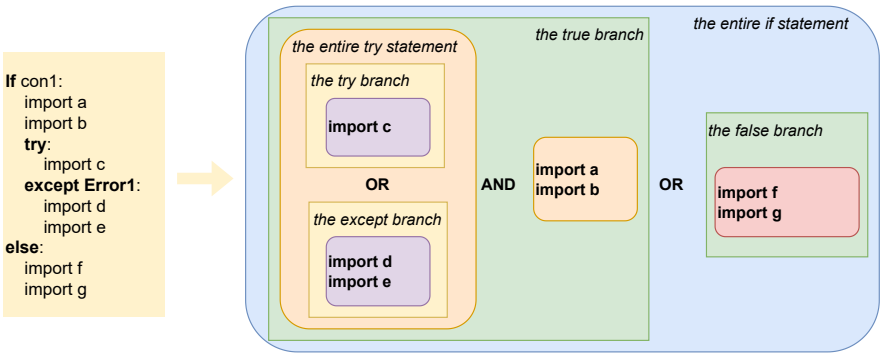


Figure 7.3: An example of block analysis for external imports.

process. The remaining import statements are regarded as external imports. PyConf executes external imports in the installed run-time environment to detect potential compatibility issues.

If the above process succeeds under Python 3, the library release enters STATUS 1. Otherwise, PyConf retries the similar process under Python 2 with some small adaptations to Python 2 syntax. The library release analyzed under Python 2 enters STATUS 2.

2) Analyze Import Blocks (STATUS 1/2 → STATUS 3). Developers may handle different run-time environments by utilizing branch statements, such as `if-else` and `try-except`, to wrap the import statements in the code. We term this practice as *multiple version control*. In such scenarios, not all imports are executed during program execution, making it essential to discern whether failures of certain imports indicate configuration issues. To address this challenge, PyConf introduces import block analysis, which effectively categorizes imports under *multiple version control* into import blocks. The main algorithm for import block analysis is detailed in Alg. 4. Additionally, Fig. 7.3 provides an illustrative example to enhance comprehension of import block analysis.

The import block analysis takes the abstract syntax tree (AST) of the current source file as input and generates two outputs: import blocks B, which are

sets of imports grouped based on the branch statements, and block-free imports D, which are import statements unaffected by any branch statements. Specifically, PyConf collects all import nodes present in the AST and verifies whether they are enclosed within branch statement nodes (line 20). Import nodes not associated with branch statements are grouped as block-free imports (line 23). For import nodes associated with branch statements, PyConf identifies the outermost branch statement node to facilitate further analysis (line 21). In the code of Fig. 7.3, all import statements are included in a `if-else` statement, so there is no block-free import.

To accommodate nested branch statements, such as the `try-except` statement within the true branch of the `if-else` statement in Fig. 7.3, PyConf adopts a recursive approach (lines 1∼17) to handle them, where the branch statement is divided into different blocks based on the branches (line 2). Each block is treated as a new virtual source file, and PyConf recursively gathers the current import blocks and block-free imports for the given branch (lines 3∼15). These current import blocks and block-free imports are then consolidated into a larger import block representative of the entire branch. This recursive process continues until all branch statements are effectively handled. The generated import blocks may exhibit nested structures due to this recursive nature. For instance, in Fig. 7.3, PyConf partitions the `if-else` statement into two blocks, highlighted in green. It then recursively handles statements in the two blocks. In the true branch block, PyConf collects all current block-free imports and the `try-except` statement as two sub-blocks, highlighted in orange. The `try-except` block is further processed to different sub-blocks, highlighted in yellow, based on the branch `try` and `except`.

7.3 Experiment Setup

In this section, we introduce the built benchmark VLibs, the baselines in the evaluation and the experiment environment.

Benchmark. We include the 5,371 libraries and their 131,720 releases that pass the three checks of PyConf in our benchmark VLibs. As PyPI libraries themselves are Python projects and have dependencies, verified PyPI libraries can form a good benchmark to evaluate the effectiveness of automatic dependency inference approaches. We show the statistics of VLibs in the last column of Table 7.1.

Baselines. We select three state-of-the-art automatic dependency inference approaches as our baselines:

Pipreqs [156]: It generates `requirements.txt` files for Python projects based on the import statements in code.

Dockerizeme [71]: It generates `Dockerfile` files for Python projects by scanning the source code. The `Dockerfile` files contain dependencies of the Python version and third-party libraries.

PyEGo [241]: It generates all information required to set up the run-time environments, including the Python version, the third-party libraries and system libraries. It utilizes knowledge graphs to store the information of PyPI libraries and invokes SMT solvers to solve the most proper version for each dependency.

Metric. We use **Pass Rate** to evaluate the performance of automatic dependency inference approaches. Pass Rate is defined as the rate of library releases whose run-time environments inferred by the approach pass all the checks of PyConf.

Environment. To avoid potential attacks on the host machine, PyConf utilizes Docker [78] to install run-time environments. We re-implement all baselines using the replicate packages provided by the authors. We conduct all experi-

ments on a Linux machine (Ubuntu 20.04 LTS) with a 112-core Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz and 256GB memory.

7.4 Result Analysis

7.4.1 Research Questions

We focus on the following research questions:

- **RQ1:** What are the configuration issues detected by PyConf?
- **RQ2:** How effective are existing automatic dependency inference approaches on VLibs?

To answer RQ1, we run PyConf on the 8,282 libraries and their 338,069 releases, as depicted in the first column of Table 7.1, to detect configuration issues. During the *Installation Check* and *Import Validation*, PyConf executes the libraries' code, capturing and logging run-time errors like `ImportError` encountered during the execution for analysis. In *Dependency Check*, PyConf collects library releases that violate the pre-defined rules in Sec. 7.2. To summarize potential configuration issues, we categorize and group the reported run-time errors based on their types. We then review the error messages to identify recurring issue patterns. Regarding RQ2, due to the time-consuming nature of building run-time environments for all baselines using the complete benchmark, we opted to sample 5,000 library releases from VLibs for analysis. To prevent potential bias during sampling, we initially select one release from each library, excluding a few that do not require configurations in VLibs, ensuring representation from all libraries. We then randomly sample the remaining releases to reach a total of 5,000 releases in the sample dataset. We run the three baselines on the sampled dataset and calculate the Pass Rates of the output configurations for each

baseline. Moreover, we conduct a comprehensive analysis to identify the primary reasons behind the failure of baselines to provide accurate configurations.

7.4.2 RQ1: Configuration Issues

Overall Results on Top Popular PyPI Libraries. We present the statistics of libraries that successfully pass the *Dependency Check* and all three checks of PyConf in the second and last columns of Table 7.1, denoted as *installed libraries* and *validated libraries*, respectively. *Installed libraries*, which are verified by PyConf along with the specified run-time environments, can be correctly set up and are available to users without encountering fatal errors. We observe that there are 7,830 (95%) installed libraries with 303,377 (90%) releases, indicating that the setup configurations of most PyPI libraries are correct. However, the situation becomes less favorable when examining the compatibility of imports in the source code with the specified run-time environments. Only 5,371 (65%) libraries, comprising 131,720 (39%) releases, successfully pass all three checks of PyConf. This indicates that approximately 30% of libraries and 51% of releases on the PyPI platform can be installed but may encounter source-level compatibility problems. Although these issues may not be severe enough to entirely prevent the usage of the library, they can adversely affect specific functionalities.

We categorize the configuration issues identified from the library releases that failed in the three checks of PyConf into three groups: *Incomplete Configuration*, *Incorrect Configuration*, and *Incorrect Code*. All these configuration issues are presented in Table 7.2. We define a configuration issue as "fatal" if it hinders the usage of the entire library release, and a configuration issue as "not fatal" if it only impacts a portion of the library's functionality. In the rest section of RQ1, we provide a detailed exploration of each configuration issue.

Incomplete Configuration. The issues under this category are raised due

Table 7.2: Configuration issues detected by PyConf. There may be multiple issues occurring in one release.

Category	Issue	Check	#Releases	Fatal?	Possible Reasons
Incomplete Configuration	Missing configuration files	Installation Check	251	✔	Missing required information
	Missing required libraries for setup	Installation Check	3,318	✔	
	Missing Python versions	Dependency Check	55,138	✘	
	Missing required libraries for direct imports	Import Validation	142,521	✘	
Incorrect Configuration	Dependency conflicts in setup	Installation Check	6,318	✔	Unsolvable constraints
	Incorrect Python versions	Installation Check	4,155	✔	Incorrect dependencies
	Other run-time Errors in setup	Installation Check	3,464	✔	Missing files
	Inconsistent configurations with metadata	Dependency Check	592	✘	Naming error
	Inconsistent version numbers with release dates	Dependency Check	12,018	✘	Confusing version orders
	Missing required modules for indirect imports	Import Validation	11,023	✘	Incorrect dependencies
	Inconsistent modules in direct imports with installed depenencies	Import Validation	6,678	✘	
	Other run-time Errors in imports	Import Validation	8,178	✘	
Incorrect Code	Missing source code	Dependency Check	2,588	✔	Creating placeholders
	Parsing error	Dependency Check	431	✔	Invalid syntax/encoding
	Multiple version control failure	Import Validation	15,507	✘	Incorrect dependencies

to the lack of some important information in the configurations. Specifically, the four issues are classified based on the missing information.

1) Missing configuration files. As mentioned in Sec. 2.2.1, most libraries use configuration files such as `requirements.txt` to record the required dependencies. However, PyConf identifies 251 library releases missing necessary configuration files, which directly results in failures in the *Installation Check*. One such instance is the installation failure of *PyAstronomy-0.10.0*, as it requires another library, `numpy`, before the successful execution of `setup.py`. However, the absence of a proper configuration file indicating the dependencies results in the

failure to install the library.

2) Missing required libraries for setup. PyConf identifies 3,318 library releases that encounter installation failures due to the absence of libraries required for setup in their configurations. This configuration issue is distinguished by the occurrence of `ModuleNotFoundError` and `ImportError` during the *Installation Check*. For example, in the library release *translators-4.0.4*, a `ModuleNotFoundError` is triggered due to a missing module `requests`. This happens when the installer tries to obtain the version from `__init__.py`, but there are some external imports that are not specified in the `setup_requires` field of `setup.py`.

3) Missing Python versions. PyConf identifies 55,138 library releases that do not indicate the required Python versions in their configurations during *Dependency Check*. The absence of specified Python versions presents significant risks to the reliability of the libraries, as the breaking changes introduced in different Python versions can impact the functionality of the libraries. A notable example is the introduction of new keywords `async` and `await` in Python version 3.5. Identifiers `async` and `await` valid in Python versions < 3.5 become invalid in Python versions > 3.5.

4) Missing required libraries for direct imports. We define **direct imports** as the import statements in the source code of the current library release, and **indirect imports** as the import statements that are called by direct imports in the source code of third-party libraries required in the configurations. PyConf identifies 142,521 library releases where modules required by direct imports are not installed because of missing corresponding library dependencies in the configurations. This issue is characterized by `ModuleNotFoundError` and `ImportError` occurring in direct imports in *Import Validation*. For example, in the library release *claripy-7.8.8.1*, there is an import statement "import celery" in the file `backends/remotetasks.py`. However, the corresponding library *celery* for the

module `celery` is not included in the configuration.

Finding 1: Developers tend to provide inadequate configurations for the usage of libraries, especially for Python versions and direct imports in source code.

Incorrect Configuration. The issues under this category are raised due to incorrect information in the configurations. Specifically, eight types of issues are classified based on incorrect information.

1) Dependency conflicts in setup. Dependency conflict in the setup occurs when the dependency constraints of third-party libraries cannot be resolved to valid versions on the PyPI platform. PyConf identifies 6,318 library releases with dependency conflicts during the *Installation Check*, as indicated by the error message "*Could not find a version that satisfies the requirement*". For instance, the library release *accountant-0.0.6* requires `enum>=1.1.5`, but the latest version of *enum* available on the PyPI platform is 0.4.7, which does not satisfy the specified constraint.

2) Incorrect Python versions. For library releases with Python version constraints, PyConf initially selects the latest Python version in the constraint for the installation and retries other Python versions in the constraints if the initial Python version fails. However, PyConf finds 4,155 library releases with Python version constraints but all the Python versions in the constraints fail in *Installation Check*. This suggests that the Python version constraints written by developers for these library releases are incorrect.

3) Other run-time errors in setup. In addition to dependency conflicts and Python version issues, PyConf identifies two types of run-time errors occurring during the setup process. Specifically, there are 966 library releases associated with `AttributeError` and 2,498 library releases associated with `FileNotFoundError` in the *Installation Check*. The `AttributeError` is caused by incorrect setup de-

pendencies, while the `FileNotFoundError` is a result of some non-configuration files being absent. As an example, the library release *aiodocker-0.1* requires `README.md`, but it does not exist.

4) Inconsistent configurations with metadata. PyConf checks the potential inconsistencies between the configurations and the library metadata. It identifies 592 library releases with such inconsistencies. The inconsistencies primarily result from the naming errors of files or folders. For example, the metadata folder in the library release *kfp-0.1.23* is named `kfp-0.1.22.dist-info`.

5) Inconsistent version numbers with release dates. When resolving version constraints of third-party libraries, `pip` installs the latest versions that meet the constraints. The selection of the latest version is determined by comparing the version number strings. However, we have discovered cases where the version number order does not align with the release date order. For example, the library *multipart* released version 2.0 in 2019 and version 0.1.1 in 2020. Developers who used this library in 2019 expected that future versions would be greater than 2.0 and thus set the constraint `multipart<0.2`. However, `pip` still considers version 0.1.1 as valid for this constraint, leading to the selection of an unexpected version. As a result, the inconsistency between the version number order and the release date order can undermine the validity of constraints set by developers. In our analysis, PyConf identifies 12,018 library releases that depend on third-party libraries with this issue.

Finding 2: Inconsistencies between version number order and release date order are prevalent in the PyPI ecosystem, undermining the validity of developers' dependency constraints.

6) Missing required modules for indirect imports. PyConf identifies 11,023 library releases where modules in indirect imports are not installed, as indicated

by `ModuleNotFoundError` and `ImportError` in *Import Validation*. This issue arises due to two possible reasons.

Firstly, the required third-party libraries may not properly handle their own dependencies. For instance, the library release *keras-bert-0.10.0* requires *keras* in the configuration and has an import statement "`import keras.backend`". However, when importing `keras.backend`, *tensorflow* is also required, but *keras* does not list it as a dependency in its configuration, resulting in import failure.

Secondly, incorrect dependencies for third-party libraries in the configurations may be the cause. For example, the library release *replit-1.4.0* has an external import statement "`import flask`", which, in turn, includes an import statement "`from markupsafe import soft_unicode`". However, `soft_unicode` is removed starting from version 2.1.0 of `markupsafe`, and there is no constraint preventing `pip` from getting the latest version of `markupsafe`, leading to the import failure.

Finding 3: Ignoring indirect dependencies is one of the major ($\sim 18\%$) incorrect configuration issues, indicating that developers often ignore indirect dependencies and only focus on the modules directly used in the source code.

7) Inconsistent modules in direct imports with installed dependencies. PyConf identifies 6,678 library releases where modules in direct imports have corresponding library dependencies in the configurations but fail to be imported. This issue arises because `pip` automatically acquires the latest available version of the required libraries, which may lead to the exclusion of certain required modules in the direct imports if they have been removed in the latest version. A prime example of this is the library *jtskit*, which is deprecated, and its developers create an empty release 0.5.0 to install another library *jsontableschema*, resulting in the failure of the direct import "import jtskit".

8) Other run-time errors in imports. We include all other run-time errors in this case. PyConf identifies 8,178 library releases with run-time errors other than `ModuleNotFoundError` and `ImportError` in the *Import Validation*, which include `TypeError`, `ValueError`, and so on. These run-time errors are induced by the execution of global statements in the module, resulting in the failure of the module import. For instance, in the library release *pandas-market-calendars-1.6.0*, there is an import statement "import trading_calendars". Upon executing this import, a global statement "NP_NAT = np.array([pd.NaT], dtype=np.int64)[0]" in the module `trading_calendars` leads to a `TypeError` due to the use of `int()` on `NaTType` type.

Finding 4: Developers make mistakes in writing configurations since 19% of configuration issues are incorrect configurations. What's more, about 50% incorrect configuration issues can only be detected by *Import Validation*, indicating the importance of source-level validation.

Incorrect Code. The issues under this category are raised due to the incorrect source code. Specifically, there are three cases classified based on source code errors.

1) Missing source code. PyConf identifies 2,588 library releases whose source code cannot be located in *Dependency Check*. PyConf cannot further validate the import statements without the source code. One possible reason we observed through manual analysis is that some library releases are published as placeholders on the PyPI platform without any actual source code. For example, the library release `mypy-protobuf-1.0` contains no source code and the `top_level.txt` file in it indicates there is no available module in the library.

2) Parsing error. PyConf identifies 431 library releases with Python files that cannot be parsed due to syntax errors, such as incorrect use of semicolons

Table 7.3: The Pass Rates (%) of three baselines on the sampled 5,000 releases from our benchmark.

Python Version?	Pipreqs	Dockerizeme	PyEGo
✔	52.9	26.6	60.7
✘	-	23.2	65.0

in Python code and encoding errors. Libraries with parsing errors cannot be handled by the Python interpreter, rendering them infeasible to be imported and used by users.

3) Multiple version control failure. In Sec. 7.2, PyConf conducts import block analysis to partition import statements in different branches into separate blocks and generate boolean expressions to validate the correctness of imports. We identify 15,507 library releases whose generated boolean expressions evaluate to *False* in *Import Validation*, indicating that none of the branches in the branch statements successfully handle the specified run-time environments. We refrain from analyzing the run-time errors in individual branches as they may not be executed in practice. Instead, we collectively refer to these cases as "multiple version control failures," highlighting the incompatibilities between the version control in the source code and the actual run-time environments.

Finding 5: Incorrect configurations can hardly be handled by the multiple version control logic in source code, as there are 5% of library releases suffering from multiple version control failures.

7.4.3 RQ2: Effectiveness of Automatic Dependency Inference Approaches

We evaluate the configurations provided by three baselines in two different settings, considering that the baseline Pipreqs does not output Python versions. In the first setting, we utilize the validated Python versions obtained in RQ1 and rely solely on the third-party library dependencies provided by the baselines to build run-time environments. In the second setting, we do not provide the validated Python versions and use those supplied by the baselines for building run-time environments. Table 7.3 presents the Pass Rates of the three baselines under these two settings. Notably, PyEGo achieves the highest Pass Rate of 65.0% when using its own inferred Python versions. This suggests that approximately 35% of library releases cannot be successfully inferred by PyEGo. On the other hand, for Pipreqs and Dockerizeme, their performance is limited, covering only 20% to 50% of library releases, despite the slight improvement when provided with the correct Python versions.

To investigate the primary reasons behind the failures of the three baselines in inferring correct dependencies, we present the major issues with at least 50 occurrences (>1%) during the check process of PyConf in Table 7.4. Surprisingly, we find that for Pipreqs and PyEGo, approximately 68% and 51% of the failures, respectively, come from dependency conflicts during setup. This suggests that some dependencies provided by these baselines are not valid on the PyPI platform. Since these baselines rely on import statements to determine which libraries should be included in the configurations, there are instances where local modules share names with third-party modules or different libraries share module names, confusing their inference. In the case of Dockerizeme, around 86% of the failures arise from missing required libraries for direct imports. This issue is also the second most common cause of failures for Pipreqs and PyEGo.

Table 7.4: The issues that three baselines fail to pass the checks of PyConf when we provide the Python versions. Only issues with more than 50 occurrences are included.

Issue	**Pipreqs**	**Dockerizeme**	**PyEGo**
Missing required libraries for setup	71	168	13
Missing required libraries for direct imports	589	3,099	597
Dependency conflicts in setup	1,675	310	823
Missing required modules for indirect imports	14	20	147
Multiple version control failure	124	15	24

One possible explanation is that the baseline databases cannot cover all libraries. For instance, PyEGo's database only includes the top 10,000 popular PyPI libraries [241], while PyPI hosts over 471 thousand libraries. Regarding the other three issues in Table 7.4, we observe that they only frequently occur in one specific baseline, suggesting that they might arise from inappropriate designs in that particular baseline's approach.

Finding 6: Current automatic dependency inference approaches fail to infer about 35% of Python projects. Most failures come from dependency conflicts and the absence of required libraries in the generated configurations.

7.5 Implications

Fewer dependency constraints lead to more configuration issues. "Less is More" seems to be a widely-used strategy to cut costs in software development. However, our findings from RQ1 reveal that 74% of configuration issues arise from insufficient dependency constraints. While these constraints may be valid and correct during the initial release of third-party libraries, they can become outdated over time as dependencies evolve. Therefore, run-time errors may occur when using certain functionalities, which cannot be detected during the setup process of run-time environments. As a result, these issues are challenging to detect without a comprehensive evaluation of the source code. Fortunately, the resolution for these issues is relatively straightforward –by adding more strict dependency constraints. We advise third-party library developers to avoid setting open constraints like `version>1.0`. Instead, they should opt for complete and strict dependency constraints that restrict Python versions and library dependencies to the verified versions at the time of release. By doing so, developers can enhance the reliability of their libraries and mitigate potential run-time errors caused by evolving dependencies.

Fewer conflict checks result in more dependency inference failures. During our analysis of why the three baselines fail to infer correct configurations, we have identified two major issues. First, some required libraries are missing, which can be resolved by updating the databases to align with the PyPI ecosystem. Second, we have observed dependency conflicts in the generated configurations. It indicates that the baselines lack sufficient conflict checks to validate the generated configurations thoroughly. For example, they do not handle the potential conflicts between the local modules inside the project and the external modules from the PyPI platform. Therefore, we recommend that future research on automatic dependency inference should incorporate more extensive conflict

checks between local projects and libraries on the PyPI platform.

7.6 Conclusion

In this chapter, we conduct an empirical study on configuration issues in the PyPI ecosystem. We propose PyConf to automatically identify configuration issues in the setup stage, the packing stage and the usage stage of third-party libraries. We also build a benchmark VLibs for the evaluation of automatic dependency inference approaches. We discover six findings and conclude two implications to facilitate the development of third-party libraries and future research on automatic dependency inference.

Chapter 8

Conclusion and Future Work

This chapter summarizes the main contributions in the thesis and presents three future directions on type inference, API recommendation, and run-time environment dependency inference, respectively.

8.1 Conclusion

In this thesis, we address significant concerns arising from the inherent challenges of dynamic type systems and runtime environments in Python software. These issues predominantly manifest as type errors and conflicts within the runtime environment, which compromise the software's reliability. To mitigate these reliability issues, we propose a suite of methods and validate their effectiveness through rigorous experiments. Our methodologies are aimed at enhancing the detection, prevention, and resolution of these reliability issues.

Specifically, in Chapter 3, we propose HiTyper, a hybrid type inference framework that iteratively integrates deep learning models and static analysis for type inference. HiTyper creates a type dependency graph for each function and validates predictions from deep learning models based on typing rules and type

rejection rules. Experiments demonstrate the effectiveness of HiTyper in type inference, enhancement for predicting rare types, and advantage of the static type inference component in HiTyper.

In Chapter 4, we present TypeGen, a few-shot generative type inference method for Python programs. Our approach incorporates static domain knowledge into language models via a novel prompt design in the *in-context* learning paradigm. Experimental results show that TypeGen outperforms both state-of-the-art supervised type inference methods and cloze-style type inference methods.

In Chapter 5, We propose a domain-aware prompt-based approach named TypeFix for repairing Python type errors. TypeFix improves prompt-based approach by incorporating domain-aware fix templates. TypeFix implements a novel fix template design to handle type errors at different levels, and mines fix templates via a novel hierarchical clustering algorithm. TypeFix incorporates domain knowledge into code prompts by applying fix templates into buggy code and invokes code pre-trained models to generate candidate patches from code prompts. Experiments demonstrate the effectiveness of TypeFix and the usefulness of fix templates mined by TypeFix.

In Chapter 6, we present an empirical study of the API recommendation task. We classify current work into query-based and code-based API recommendation, and build a benchmark named APIBench to align the performance of different recommendation approaches. We conclude some findings based on the empirical results of current approaches.

For query-based API recommendation approaches, we find that 1) recommending method-level APIs is still challenging; 2) query reformulation techniques have great potential to improve the quality of user queries thus they can help current approaches better recommend APIs. What's more, user queries also contain some meaningless and verbose words and even a simple word deletion method can

improve the performance; 3) approaches built upon different data sources have quite different performances. Q&A forums such as Stack Overflow can greatly help mitigate the gap between user queries and API descriptions.

For code-based API recommendation, we emphasize the superior performance of current deep learning models such as Transformer. However, they still face the challenge of recommending user-defined APIs. We also find different contexts, such as different location of recommendation points and context length, can impact the performance of current approaches. Besides, current approaches suffer from recommending cross-domain APIs.

In Chapter 7, we conduct an empirical study on configuration issues in the PyPI ecosystem. We propose PyConf to automatically identify configuration issues in the setup stage, the packing stage and the usage stage of third-party libraries. We also build a benchmark VLibs for the evaluation of automatic dependency inference approaches. We discover six findings and conclude two implications to facilitate the development of third-party libraries and future research on automatic dependency inference.

8.2 Future Work

8.2.1 Synergistic Type Inference

Type information plays a crucial role in the development and maintenance of scalable software. The absence of type information in code introduces vulnerabilities and complicates maintenance tasks. Dynamic languages, which often omit type information to accelerate prototyping, require robust type inference mechanisms to enhance software reliability. Traditional static type inference methods are sound, however, they are limited in their ability to infer types across a broad range of variables. On the other hand, data-driven techniques, particularly those

utilizing recent advancements in Large Language Models (LLMs), have shown significant effectiveness in predicting types. Nevertheless, these methods do not provide assurances regarding the accuracy of their predictions. Consequently, existing methodologies struggle to simultaneously deliver high performance and ensure prediction correctness in type inference.

In this thesis, we introduce HiTyper, which employs static analysis techniques to enhance the accuracy of type predictions. Additionally, we propose TypeGen, a generative type inference approach that leverages the capabilities of large language models (LLMs) to deliver interpretable type predictions. Together, these two methodologies address some of the limitations inherent in existing type inference methods. Despite these advancements, a significant challenge remains: LLMs do not inherently learn from their incorrect type predictions, which curtails their performance.

To overcome this limitation, we propose a future direction of synergistic type inference, designed to foster collaboration between static type checkers and LLMs. In this model, static type checkers and LLMs function as cooperative agents. The static checkers validate the correctness of type predictions, while the LLMs utilize the feedback from error messages to learn and adjust incorrect predictions. We believe this cooperative framework can substantially enhance the effectiveness of LLMs in type inference.

8.2.2 High-quality API Recommendation

Based on the insights garnered from Chapter 6, we outline several prospective avenues for enhancing API recommendation systems. For query-based approaches, we advocate for the integration of query reformulation techniques with existing API recommendation frameworks to improve their performance. However, determining the optimal query reformulation strategy remains an open area

for future research. Additionally, we propose the adoption of few-shot learning methods and the utilization of diverse data sources to effectively bridge the discrepancies between user queries and the knowledge base, particularly in low-resource settings.

For code-based API recommendation systems, we recommend that future research should concentrate on refining the accuracy and relevance of recommendations for user-defined APIs. Moreover, it is advisable to extend training methodologies across multiple domains rather than restricting them to a single domain. This multi-domain approach could potentially enhance the robustness and applicability of API recommendation systems, thereby contributing to more personalized and effective developer support.

Additionally, we identify further research directions in the evaluation of API recommendation. Firstly, while this thesis primarily concentrates on benchmarking and delivering an objective evaluation of various API recommendation methods, it does not assess the quality of summaries provided by some approaches for the recommended APIs. To bridge this gap, subjective evaluations, such as developer surveys, could be employed to ascertain the quality of these API descriptions and their usage information. This would provide a more comprehensive understanding of the utility and clarity of the API recommendations from a developer's perspective.

Secondly, our empirical findings highlight the effectiveness of query reformulation techniques in enhancing query quality. Although our focus was predominantly on assessing the performance of API recommendation systems, a detailed exploration of different query reformulation strategies was not conducted. Future research could therefore delve deeper into these techniques, examining their role and effectiveness in facilitating downstream tasks. Such studies would likely yield valuable insights into optimizing query reformulation for improved API recom-

mendation outcomes.

8.2.3 Source-level Run-time Environment Dependency Inference

In Chapter 7, we introduce PyConf, a novel approach for detecting conflicts in source-level runtime environments, and we present a comprehensive empirical study on configuration issues identified through this methodology. While we have managed to discern specific patterns and underlying causes of these issues through manual analysis, currently, there lacks a method to correct these incorrect configurations provided by developers. We believe that source-level runtime environment dependency inference could represent a significant advancement in both preventing and resolving the configuration issues detected by PyConf.

This method would consider not only the interrelationships among various third-party dependencies but also assess the compatibility of external APIs utilized within the software with these third-party libraries. Given the dynamic and evolving nature of third-party dependencies, a static configuration file established by developers may become outdated or incompatible over time. We believe that a more effective strategy for setting up Python runtime environments would involve employing a source-level runtime environment dependency inference framework to generate the most recent compatible configurations automatically. The runtime environment would then be constructed based on these dynamically generated, up-to-date configurations.

Appendix A

List of Publications

1. **Yun Peng**, Ruida Hu, Ruoke Wang, Cuiyun Gao, Shuqing Li, Michael R. Lyu. *Less is More? An Empirical Study on Configuration Issues in Python PyPI Ecosystem.* In Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024.

2. **Yun Peng**, Shuzheng Gao, Cuiyun Gao, Yintong Huo, Michael R. Lyu. *Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors.* In Proceedings of the 46th International Conference on Software Engineering (ICSE), 2024.

3. **Yun Peng**, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, Michael R. Lyu. *Generative Type Inference for Python.* In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. **(ACM SIGSOFT Distinguished Paper Award)**

4. Shuqing Li, Cuiyun Gao, Jianping Zhang, Yujia Zhang, Yepang Liu, Jiazhen Gu, **Yun Peng**, Michael R. Lyu. *Less Cybersickness, Please: Demystifying and Detecting Stereoscopic Visual Inconsistencies in Extended Reality Applications.* In Proceedings of the ACM International Conference

on the Foundations of Software Engineering (FSE), 2024.

5. Yichen Li, **Yun Peng**, Yintong Huo, Michael R. Lyu. *Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context.* In Proceedings of the First International Workshop on Large Language Models for Code (LLM4Code), 2024.

6. Chaozheng Wang, Zongjie Li, **Yun Peng**, Shuzheng Gao, Sirong Chen, Shuai Wang, Cuiyun Gao, Michael R. Lyu. *REEF: A Framework for Collecting Real-World Vulnerabilities and Fixes.* In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023. (Industry Challenge Track) **(Distinguished Paper Award of Industry Challenge Track)**

7. Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, **Yun Peng**, Hongyu Zhang, Michael R. Lyu. *Prompt Tuning in Code Intelligence: An Experimental Evaluation.* In IEEE Transactions on Software Engineering (TSE), 2023.

8. Yujia Chen, Cuiyun Gao, Xiaoxue Ren, **Yun Peng**, Xin Xia, Michael Lyu. *API Usage Recommendation via Multi-View Heterogeneous Graph Representation Learning.* In IEEE Transactions on Software Engineering (TSE), 2023. (Journal First)

9. **Yun Peng**, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, Michael Lyu. *Revisiting, Benchmarking and Exploring API Recommendation: How Far Are We?.* In IEEE Transactions on Software Engineering (TSE), 2022. (Journal First)

10. Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, **Yun Peng**, Hongyu Zhang, Michael R. Lyu. *No More Fine-tuning? An Experimental Evaluation of Prompt Tuning in Code Intelligence.* In Proceedings of the 30th ACM

Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022.

11. **Yun Peng**, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, Michael R. Lyu. *Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python.* In Proceedings of the 44th International Conference on Software Engineering (ICSE), 2022. **(ACM SIGSOFT Distinguished Paper Award Nomination)**

Reference

[1] Geeks4geeks website. https://www.geeksforgeeks.org/, 2021. 141

[2] Java2s website. http://www.java2s.com/, 2021. 141

[3] Kode java website. https://kodejava.org/, 2021. 141

[4] Mohammad Masudur Rahman 0001 and Chanchal K. Roy. Improved query reformulation for concept location using coderank and document structures. PeerJ PrePrints, 5, 2017. 133

[5] AIDanial. Count lines of code. https://github.com/AlDanial/cloc, 2021. 143

[6] aldur. The return value at line 295., 2021. https://github.com/aldur/MatasanoCrypto/blob/master/matasano/blocks.py. 61

[7] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, page 91–105, New York, NY, USA, 2020. Association for Computing Machinery. 18, 26, 34, 35, 36, 39, 53, 55, 56, 65, 80, 81

[8] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: neural type hints. In Alastair F. Donaldson and Emina Torlak, editors, Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020, pages 91–105. ACM, 2020. 95

[9] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019. 144

[10] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05, page 428–452, Berlin, Heidelberg, 2005. Springer-Verlag. 25, 34, 65

[11] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Why do software packages conflict? In Michele Lanza, Massimiliano Di Penta, and Tao Xie, editors, 9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland, pages 141–150. IEEE Computer Society, 2012. 31, 190

[12] Sid Black, Stella Biderman, Leo Gao, Phil Wang, and Connor Leahy. Gpt-neo, 2022. https://github.com/EleutherAI/gpt-neo. 80

[13] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. Comput. Networks, 30(1-7):107–117, 1998. 28

[14] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler,

Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020. 80

[15] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pages 213–222, 2009. 127

[16] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. Automated query reformulation for efficient search based on query logs from stack overflow. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 1273–1285. IEEE, 2021. 133, 141, 146, 147, 156, 166

[17] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. The demo link for sequer. https://sequer-tpznovfjxa-uc.a.run.app/?query=, 2021. 146

[18] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching connected api subgraph via text phrases. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pages 1–11, 2012. 28

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde

de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. CoRR, abs/2107.03374, 2021. 97, 98, 117

[20] Sheng Chen and Martin Erwig. Principal type inference for gadts. In Rastislav Bodík and Rupak Majumdar, editors, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 416–428. ACM, 2016. 25, 65

[21] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Trans. Software Eng., 47(9):1943–1959, 2021. 27

[22] Wei Cheng, Xiangrong Zhu, and Wei Hu. Conflict-aware inference of python compatible runtime environments with domain knowledge graph. In 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE

2022, Pittsburgh, PA, USA, May 25-27, 2022, pages 451–461. ACM, 2022. 32, 191, 194

[23] Robert Collins. Pep 508 –dependency specification for python software packages, 2015. https://peps.python.org/pep-0508/. 195

[24] Mona Diab. Data paucity and low resource scenarios: Challenges and opportunities. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '20, page 3612, New York, NY, USA, 2020. Association for Computing Machinery. 182

[25] Django. Django api reference. https://docs.djangoproject.com/en/3.2/ref/, 2021. 138

[26] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. A survey on in-context learning, 2023. 68

[27] Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. CoRR, abs/2105.09352, 2021. 28, 95

[28] Andrea Renika D' Souza, Di Yang, and Cristina V Lopes. Collective intelligence for smarter api recommendations in python. In 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 51–60. IEEE, 2016. 30, 136

[29] Michael Emmi and Constantin Enea. Symbolic abstract data type inference. In Rastislav Bodík and Rupak Majumdar, editors, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 513–525. ACM, 2016. 25, 65

[30] Stack Exchange. Stack exchange data dump. https://archive.org/details/stackexchange, 2021. 139

[31] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. 26, 66

[32] Flask. Flask api reference. https://flask.palletsprojects.com/en/2.0.x/api/, 2021. 138

[33] J. Fleiss. Measuring nominal scale agreement among many raters. Psychological Bulletin, 76:378–382, 1971. 141

[34] Apache Software Foundation. Lucene. https://lucene.apache.org/, 2021. 142, 147, 148

[35] Eclipse Foundation. The eclipse ide. https://www.eclipse.org/eclipseide/, 2021. 149

[36] Python Software Foundation. Official documentation of Mypy, 2020. https://mypy.readthedocs.io/en/stable/builtin_types.html. 17

[37] Python Software Foundation. Official documentation of Python3, 2020. https://docs.python.org/3. 17, 56

[38] Python Software Foundation. Cpython. python's official implementation, 2021. https://github.com/python/cpython. 51

[39] Python Software Foundation. The pip tool, 2023. https://pypi.org/project/pip/. 22, 190

[40] Python Software Foundation. The python ast module, 2023. https://docs.python.org/3/library/ast.html. 101, 198

[41] Python Software Foundation. The python package index, 2023. https://pypi.org/. 2, 22, 31, 190, 194

[42] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, pages 254–265, 2016. 30, 149

[43] Jaroslav M. Fowkes and Charles Sutton. Parameter-free probabilistic API mining across github. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, pages 254–265. ACM, 2016. 144, 150

[44] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net, 2023. 27, 66, 80

[45] Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. Pfrl: a pytorch-based deep reinforcement learning library, 2023. https://github.com/pfnet/pfrl. 191

[46] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, page 1859–1866, New York, NY, USA, 2009. Association for Computing Machinery. 25, 34, 65

[47] Philip Gage. A new algorithm for data compression. C Users J., 12(2):23–38, February 1994. 54

[48] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for C programs. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pages 459–470. IEEE Computer Society, 2015. 27

[49] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pages 1933–1945, 2023. 92

[50] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R. Lyu. Constructing effective in-context demonstration for code intelligence tasks: An empirical study. CoRR, abs/2304.07575, 2023. 77

[51] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. Crash-avoiding program repair. In Dongmei Zhang and Anders Møller, editors, Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, pages 8–18. ACM, 2019. 27

[52] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. IEEE Trans. Software Eng., 45(1):34–67, 2019. 27

[53] gengo. The return value at line 853., 2021. https://github.com/gengo/memsource-wrap/blob/master/memsource/api.py. 61

[54] Loukas Georgiadis, Robert Tarjan, and Renato Werneck. Finding dominators in practice. volume 10, pages 69–94, 01 2006. 52

[55] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In Dongmei Zhang and Anders Møller, editors, Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, pages 19–30. ACM, 2019. 27

[56] Inc. GitHub. Github octoverse report on programming languages, 2022. https://octoverse.github.com/2022/top-programming-languages. 1, 34, 95, 190

[57] Google. Android api reference. https://developer.android.com/reference, 2021. 138

[58] Google. The google news word2vec model. https://code.google.com/archive/p/word2vec/, 2021. 154

[59] Google. The interface of google prediction service. http://suggestqueries.google.com/complete/search?, 2021. 146, 147

[60] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 631–642, 2016. 28, 29, 127, 134, 147, 148

[61] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, Proceedings of the 60th Annual Meeting of the Association for

Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022, pages 7212–7225. Association for Computational Linguistics, 2022. 26, 66, 80

[62] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021. 26, 66

[63] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. Empirical Software Engineering, 19, 10 2013. 34

[64] Foyzul Hassan and Xiaoyin Wang. Hirebuild: an automatic approach to history-driven repair of build scripts. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pages 1078–1089. ACM, 2018. 31

[65] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. Maxsmt-based type inference for python 3. In Hana Chockler and Georg Weissenbacher, editors, Computer Aided Verification, pages 12–19, Cham, 2018. Springer International Publishing. 34

[66] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. Pyart: Python api recommendation in real-time. In arXiv, 2021. 31, 136, 137, 144, 149, 150

[67] Michael A. Hedderich, Lukas Lange, Heike Adel, Jannik Strötgen, and Dietrich Klakow. A survey on recent approaches for natural language processing in low-resource scenarios, 2021. 182

[68] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery. 34, 51, 52, 65

[69] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, pages 837–847. IEEE Computer Society, 2012. 30, 136

[70] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. SAVER: scalable, precise, and safe memory-error repair. In Gregg Rothermel and Doo-Hwan Bae, editors, ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, pages 271–283. ACM, 2020. 27

[71] Eric Horton and Chris Parnin. Dockerizeme: automatic inference of environment dependencies for python code snippets. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pages 328–338. IEEE / ACM, 2019. 32, 191, 193, 194, 203

[72] Eric Horton and Chris Parnin. V2: fast detection of configuration drift

in python. In 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, pages 477–488. IEEE, 2019. 32

[73] Mingzhe Hu, Yu Zhang, Wenchao Huang, and Yan Xiong. Static type inference for foreign functions of python. In 32nd International Symposium on Software Reliability Engineering, October 2021. 49

[74] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, pages 388–398. IEEE, 2019. 27

[75] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. Interactive, effort-aware library version harmonization. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020, pages 518–529. ACM, 2020. 31, 32

[76] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 293–304. IEEE, 2018. 28, 29, 127, 134, 137, 141, 142, 147, 148, 150, 169, 182

[77] HuggingFace. Huggingface hub, 2023. https://huggingface.co/. 81

[78] Docker Inc. Docker, 2023. https://www.docker.com/. 203

[79] IntelLabs. The return value at line 95. https://github.com/IntelLabs/coach/blob/master/rl_coach/memories/non_episodic/experience_replay.py, 2021. 61

[80] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Proceedings of the 16th International Symposium on Static Analysis, SAS '09, page 238–255, Berlin, Heidelberg, 2009. Springer-Verlag. 25, 34, 65

[81] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. Learning type annotation: Is big data enough? In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, page 1483–1486, New York, NY, USA, 2021. Association for Computing Machinery. 26, 65, 79, 81

[82] Jetbrains. Python developer survey conducted by jetbrains and python software foundation, 2020. https://www.jetbrains.com/lp/python-developers-survey-2020/. 34, 138, 186

[83] JetBrains. The intellij idea ide. https://www.jetbrains.com/idea/, 2021. 149

[84] JetBrains. The pycharm ide. https://www.jetbrains.com/pycharm/, 2021. 149

[85] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. Inferring program transformations from singular examples via big code. In 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, pages 255–266. IEEE, 2019. 27

[86] Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: code-aware neural machine translation for automatic program repair. In 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021, pages 1161–1173. IEEE, 2021. 28, 95, 117, 118

[87] Elise Jing, Kristiana Schneck, Dennis Egan, and Scott A. Waterman. Identifying introductions in podcast episodes from automatically generated transcripts, 2021. 146

[88] Bingyi Kang, Saining Xie, Marcus Rohrbach, Zhicheng Yan, Albert Gordo, Jiashi Feng, and Yannis Kalantidis. Decoupling representation and classifier for long-tailed recognition, 2020. 35

[89] C. M. Khaled Saifullah, M. Asaduzzaman, and C. K. Roy. Exploring type inference techniques of dynamically typed languages. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 70–80, 2020. 34

[90] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 150–162. IEEE, 2021. 31, 127, 136, 137, 144, 149, 150

[91] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. The replication package of travtrans. https://github.com/facebookresearch/code-prediction-transformer, 2021. 182

[92] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In 2nd LLVM Workshop on the LLVM Compiler Infrastructure in HPC. 34

[93] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of api evolution literature. ACM Comput. Surv., 54(8), oct 2021. 137

[94] lanwuwei. A pre-trained bert on stackoverflow corpus. https://github.com/lanwuwei/BERTOverflow, 2021. 148, 154

[95] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. ACM Comput. Surv., 53(3), June 2020. 34

[96] learner. how to solve module 'gym.wrappers' has no attribute 'monitor'?, 2022. https://stackoverflow.com/questions/71411045/how-to-solve-module-gym-wrappers-has-no-attribute-monitor. 192

[97] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. ICSE '19, page 795–806. IEEE Press, 2019. 144

[98] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Memfix: static analysis-based repair of memory deallocation errors for C. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pages 95–106. ACM, 2018. 27

[99] Jukka Lehtosalo. PEP 589 – TypedDict: Type hints for dictionaries with a fixed set of keys, March 2019. https://www.python.org/dev/peps/pep-0589/. 26, 34, 65, 95

[100] Ivan Levkivskyi, Jukka Lehtosalo, and Łukasz Langa. PEP 544 – protocols: Structural subtyping (static duck typing), March 2017. https://www.python.org/dev/peps/pep-0544/. 26, 34, 65, 95

[101] Bohan Li, Yutai Hou, and Wanxiang Che. Data augmentation approaches in natural language processing: A survey. AI Open, 2022. 156

[102] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. Editsum: A retrieve-and-edit framework for source code summarization. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 155–166, 2021. 77

[103] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. Towards enhancing in-context learning for code generation. CoRR, abs/2303.17780, 2023. 82

[104] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: context-based code transformation learning for automated program repair. In Gregg Rothermel and Doo-Hwan Bae, editors, ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, pages 602–614. ACM, 2020. 27

[105] libraries.io. libraries.io, 2023. https://libraries.io/. 73

[106] Chun-Yang Ling, Yan-Zhen Zou, Ze-Qi Lin, and Bing Xie. Graph embedding based api graph search and recommendation. Journal of Computer Science and Technology, 34(5):993–1006, 2019. 28, 29

[107] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? In Workshop on Knowledge Extraction and Integration for Deep Learning Architectures; Deep Learning Inside Out, 2021. 78

[108] Jialun Liu, Yifan Sun, Chuchu Han, Zhaopeng Dou, and Wenhui Li. Deep representation learning on long-tailed data: A learnable embedding augmentation perspective, 2020. 35

[109] Mingwei Liu, Xin Peng, Andrian Ma rcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. Generating query-specific class api summaries. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 120–130, 2019. 28, 29, 127, 134, 137, 147, 148, 183

[110] Xiaoyu Liu, LiGuo Huang, and Vincent Ng. Effective api recommendation without historical software repositories. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 282–292, 2018. 30

[111] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, pages 545–549. IEEE Computer Society, 2015. 133

[112] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In Sarfraz Khurshid and Corina S. Pasareanu, editors, ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, pages 101–114. ACM, 2020. 28, 95, 98, 117, 118

[113] Edward Ma. Nlp augmentation. https://github.com/makcedward/nlpaug, 2019. 146, 147, 154

[114] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In Proceedings of the 41st International Conference on Software Engineering, ICSE '19, page 304–315. IEEE Press, 2019. 34

[115] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021, pages 505–509. IEEE, 2021. 28

[116] Matplotlib. Matplotlib api reference. https://matplotlib.org/stable/api/index.html, 2021. 138

[117] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In Proceedings of the 33rd International Conference on Software Engineering, pages 111–120, 2011. 28

[118] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, pages 691–701. ACM, 2016. 27

[119] Meta. Llama3 blog. https://ai.meta.com/blog/meta-llama-3/, 2024. 27

[120] Microsoft. Github website. https://github.com/, 2021. 143

[121] Microsoft. The visual studio code editor. https://code.visualstudio.com/, 2021. 149

[122] Microsoft. Copilot, 2023. https://github.com/features/copilot. 117

[123] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc. 54

[124] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States, pages 3111–3119, 2013. 56

[125] George A. Miller. Wordnet: A lexical database for english. Commun. ACM, 38(11):39–41, November 1995. 154

[126] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022, pages 11048–11064. Association for Computational Linguistics, 2022. 90

[127] Amir M. Mir, Evaldas Latoskinas, and Georgios Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type infer-

ence. In 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021, pages 585–589. IEEE, 2021. 35, 55, 67, 78

[128] Amir M Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py, 2022. https://github.com/saltudelft/type4py. 65, 67

[129] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pages 2241–2252. ACM, 2022. 18, 26, 34, 36, 51, 55, 56, 78, 79, 80, 81

[130] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pages 2241–2252. ACM, 2022. 95

[131] Martin Monperrus. Automatic software repair: A bibliography. ACM Comput. Surv., 51(1):17:1–17:24, 2018. 27

[132] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. Empirical Software Engineering, 17(6):703–737, 2012. 28

[133] Mypy. https://github.com/python/mypy/. 18, 20, 25, 49, 65, 95

[134] Benjamin Newman, Prafulla Kumar Choubey, and Nazneen Rajani. P-adapters: Robustly extracting factual information from language models with diverse prompts, 2021. 146

[135] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 511–522, 2016. 30, 127

[136] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, pages 858–868. IEEE, 2015. 31, 136

[137] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In 2012 34th International Conference on Software Engineering (ICSE), pages 69–79. IEEE, 2012. 136

[138] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pages 772–781. IEEE Computer Society, 2013. 27

[139] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. Focus: A recommender system for mining api function calls and usage patterns. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 1050–1060. IEEE, 2019. 30, 136, 144, 149, 150, 175

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

guage model for code with multi-turn program synthesis. In The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net, 2023. 80

[141] Haoran Niu, Iman Keivanloo, and Ying Zou. Api usage pattern recommendation for software development. Journal of Systems and Software, 129:127–139, 2017. 30

[142] Numpy. Numpy api reference. https://numpy.org/doc/stable/reference/, 2021. 138

[143] Wonseok Oh and Hakjoo Oh. Pyter: effective program repair for python type errors. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, pages 922–934. ACM, 2022. 3, 27, 65, 95, 96, 97, 98, 116, 117, 118

[144] The open source community. The pypi web page of library pipreqs 0.4.13., 2023. https://pypi.org/project/pipreqs/. 196

[145] OpenAI. Chatgpt, 2022. https://openai.com/blog/chatgpt. 80

[146] OpenAI. Codex api reference, 2023. https://platform.openai.com/docs/api-reference/completions/create. 118

[147] OpenAI. GPT-4 technical report. CoRR, abs/2303.08774, 2023. 27

[148] Oracle. Java se 8 api documentation. www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html, 2021. 127, 138

[149] Pandas. Pandas api reference. https://pandas.pydata.org/docs/reference/index.html, 2021. 138

[150] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. Conflictjs: finding and understanding conflicts between javascript libraries. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pages 741–751. ACM, 2018. 31, 190

[151] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. Data flow refinement type inference. Proc. ACM Program. Lang., 5(POPL):1–31, 2021. 25, 65

[152] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Hityper, 2022. https://github.com/JohnnyPeng18/HiTyper. 25, 66

[153] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In Proceedings of the 44th International Conference on Software Engineering, ICSE '22, page 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery. 26, 65, 70, 73, 78, 79, 80, 81

[154] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael R. Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pages 2019–2030. ACM, 2022. 95

[155] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael R. Lyu. Domain knowledge matters: Improving prompts with fix templates for repairing python type errors. CoRR, abs/2306.01394, 2023. 65

[156] pipreqs. pipreqs, 2023. https://github.com/bndr/pipreqs. 32, 193, 203

[157] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, page 209–220, New York, NY, USA, 2020. Association for Computing Machinery. 26, 65, 79, 81

[158] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural Type Prediction with Search-Based Validation, page 209–220. Association for Computing Machinery, New York, NY, USA, 2020. 34, 35, 36, 52, 55

[159] Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can openai's codex fix bugs?: An evaluation on quixbugs. In 3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022, pages 69–75. IEEE, 2022. 28

[160] Pyre check. https://pyre-check.org/. 18, 25, 34, 36, 49, 55, 59, 65

[161] Pyright. https://github.com/microsoft/pyright. 25, 65

[162] Pysonar2. https://github.com/yinwang0/pysonar2. 34

[163] Python. The python ast module, 2021. https://github.com/python/cpython/blob/3.9/Lib/ast.py. 46

[164] Python. Python standard library. https://docs.python.org/3/library/, 2021. 138

[165] Python. The typeshed project, 2021. https://github.com/python/typeshed. 49

[166] Pytype. https://github.com/google/pytype. 18, 25, 34, 36, 49, 53, 55, 59, 65

[167] Lianyong Qi, Qiang He, Feifei Chen, Wanchun Dou, Shaohua Wan, Xuyun Zhang, and Xiaolong Xu. Finding all you need: web apis recommendation in web of things through keywords search. IEEE Transactions on Computational Social Systems, 6(5):1063–1072, 2019. 28

[168] Maithra Raghu and Eric Schmidt. A survey of deep learning for scientific discovery, 2020. 146

[169] Mohammad Masudur Rahman and Chanchal Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. 06 2018. 134

[170] Mohammad Masudur Rahman and Chanchal Roy. Nlp2api: Query reformulation for code search using crowdsourced knowledge and extra-large data analytics. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 714–714, 2018. 127, 133, 134, 137, 146, 147

[171] Mohammad Masudur Rahman and Chanchal Roy. The replication package for nlp2api. https://github.com/masud-technope/NLP2API-Replication-Package, 2021. 146

[172] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. Rack: Automatic api recommendation using crowdsourced knowledge. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 349–359. IEEE, 2016. 28, 29, 127, 133, 134, 137, 147, 148

[173] Vikas Raunak, Siddharth Dalmia, Vivek Gupta, and Florian Metze. On long-tailed phenomena in neural machine translation. In Findings of the Association for Computational Linguistics: EMNLP 2020, pages 3088–3095, Online, November 2020. Association for Computational Linguistics. 35

[174] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. Commun. ACM, 60(10):91–100, September 2017. 34

[175] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, page 731–747, New York, NY, USA, 2016. Association for Computing Machinery. 31, 136, 149, 150

[176] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". SIGPLAN Not., 50(1):111–124, January 2015. 34

[177] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 419–428, 2014. 31, 127, 136, 144, 150

[178] Jiawei Ren, Cunjun Yu, Shunan Sheng, Xiao Ma, Haiyu Zhao, Shuai Yi, and Hongsheng Li. Balanced meta-softmax for long-tailed visual recognition, 2020. 35

[179] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to Recommender Systems Handbook, pages 1–35. Springer US, Boston, MA, 2011. 136

[180] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery. 34

[181] Romain Robbes and Michele Lanza. Improving code completion with program history. Automated Software Engineering, 17(2):181–212, 2010. 127

[182] Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. Found. Trends Inf. Retr., 3(4):333–389, 2009. 77

[183] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. IEEE Transactions on Software Engineering, 39(5):613–637, 2013. 137

[184] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. Recommendation Systems in Software Engineering. Springer Publishing Company, Incorporated, 2014. 137

[185] Salesforce. The codet5-base model, 2021. https://huggingface.co/Salesforce/codet5-base. 117

[186] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python.

In 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021, pages 1646–1657. IEEE, 2021. 39, 49

[187] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Effective smart completion for javascript. Technical Report RC25359, 2013. 30

[188] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11):2673–2681, 1997. 25

[189] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. 54

[190] sernst. The return value at line 35., 2021. https://github.com/sernst/cauldron/blob/master/cauldron/steptest/functional.py. 61

[191] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and structuring user queries to support efficient free-form code search. In Proceedings of the 40th International Conference on Software Engineering, ICSE '18, page 945, New York, NY, USA, 2018. Association for Computing Machinery. 133

[192] snyk. Jvm ecosystem report 2020. https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-i 2021. 138, 186

[193] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais,

and Benoit Baudry. The emergence of software diversity in maven central. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada, pages 333–343. IEEE / ACM, 2019. 31, 190

[194] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: improving configuration management with operating system causality analysis. In Thomas C. Bressoud and M. Frans Kaashoek, editors, Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, pages 237–250. ACM, 2007. 31

[195] Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. Static type recommendation for python. In 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022, pages 98:1–98:13. ACM, 2022. 95

[196] Xiaobing Sun, Congying Xu, Bin Li, Yucong Duan, and Xintong Lu. Enabling feature location for api method recommendation and usage location. IEEE Access, 7:49872–49881, 2019. 28

[197] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada, pages 3104–3112, 2014. 27

[198] ThePeshMod. "from gym.wrappers import monitor" has been deprecated, 2022. https://github.com/pfnet/pfrl/issues/172. 192

[199] Ferdian Thung, Shaowei Wang, David Lo, and Julia L. Lawall. Automatic recommendation of API methods from feature requests. In 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, pages 290–300. IEEE, 2013. 28, 29

[200] Inc Tidelift. Libraries.io - the open source discovery service, 2023. https://libraries.io/. 195

[201] tomdean. The return value at line 56., 2021. https://github.com/tomdean/growser/blob/master/growser/handlers/rankings.py. 61

[202] Open Source Tool. pipdeptree, 2023. https://github.com/tox-dev/pipdeptree. 198

[203] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 269–280, 2014. 30, 136

[204] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. PEP 484 – Type Hints, 2014. https://www.python.org/dev/peps/pep-0484/. 26, 34, 65, 95

[205] Efstathiou Vasiliki, Chatzilenas Christos, and Spinellis Diomidis. Word Embeddings for the Software Engineering Domain, March 2018. 154

[206] Ben Wang and Aran Komatsuzaki. Gpt-j, 2022. https://github.com/kingoflolz/mesh-transformer-jax. 80

[207] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. smartpip: A smart approach to resolving python dependency conflict issues. In 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022, pages 93:1–93:12. ACM, 2022. 31

[208] Jiawei Wang, Li Li, and Andreas Zeller. Restoring execution environments of jupyter notebooks. In 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021, pages 1622–1633. IEEE, 2021. 32

[209] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In 2013 10th Working Conference on Mining Software Repositories (MSR), pages 319–328. IEEE, 2013. 30

[210] Wenxuan Wang, Wenxiang Jiao, Yongchang Hao, Xing Wang, Shuming Shi, Zhaopeng Tu, and Michael R. Lyu. Understanding and improving sequence-to-sequence pretraining for neural machine translation. In Annual Meeting of the Association for Computational Linguistics, 2022. 27

[211] Wenxuan Wang and Zhaopeng Tu. Rethinking the value of transformer components. In International Conference on Computational Linguistics, 2020. 27

[212] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. HERO: on the chaos when PATH meets modules. In 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021, pages 99–111. IEEE, 2021. 31, 32

[213] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. Watchman: monitoring dependency conflicts for python library ecosystem. In Gregg Rothermel and Doo-Hwan Bae, editors, ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, pages 125–135. ACM, 2020. 31, 190

[214] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pages 319–330. ACM, 2018. 31, 190

[215] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. Could I have a stack trace to examine the dependency conflict issue? In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pages 572–583. IEEE / ACM, 2019. 31, 190

[216] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. Will dependency conflicts affect my program's semantics? IEEE Trans. Software Eng., 48(7):2295–2316, 2022. 31

[217] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code un-

derstanding and generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. 26, 66, 80, 97, 100, 115

[218] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. CoRR, abs/2201.11903, 2022. 67, 75, 76

[219] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. In 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net, 2020. 26

[220] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. CoRR, abs/2005.02161, 2020. 65

[221] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: interactive system configuration repair. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pages 625–636. IEEE Computer Society, 2017. 31

[222] Fengcai Wen, Emad Aghajani, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Siri, write the next method. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 138–149. IEEE, 2021. 30, 136

[223] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pages 1–11. ACM, 2018. 27

[224] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020, pages 1556–1560. ACM, 2020. 97, 116

[225] wtsi hgi. The return value at line 151., 2021. https://github.com/wtsi-hgi/metadata-check/blob/master/mcheck/metadata/seqscape_metadata/seqscape_metadata.py. 61

[226] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models. CoRR, abs/2210.14179, 2022. 81, 118

[227] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pages 1482–1494. IEEE, 2023. 28

[228] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, pages 959–971. ACM, 2022. 28, 95, 96, 98, 117, 118, 122

[229] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. CoRR, abs/2301.13246, 2023. 81

[230] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. CoRR, abs/2304.00385, 2023. 28

[231] Rensong Xie, Xianglong Kong, Lulu Wang, Ying Zhou, and Bixin Li. Hirec: Api recommendation using hierarchical context. In 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pages 369–379. IEEE, 2019. 135, 136

[232] Wei Xiong, Zhihui Lu, Bing Li, Bo Hang, and Zhao Wu. Automating smart recommendation from natural language api descriptions via representation learning. Future Generation Computer Systems, 87:382–391, 2018. 28, 29

[233] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pages 416–426. IEEE / ACM, 2017. 27

[234] Congying Xu, Xiaobing Sun, Bin Li, Xintong Lu, and Hongjing Guo. MULAPI: improving API method recommendation with API usage location. J. Syst. Softw., 142:195–205, 2018. 28, 29

[235] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, page 607–618, New York, NY, USA, 2016. Association for Computing Machinery. 34

[236] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. Software Eng., 43(1):34–55, 2017. 27

[237] Le Xue, Mingfei Gao, Zeyuan Chen, Caiming Xiong, and Ran Xu. Robustness evaluation of transformer-based form field extractors via form attacks, 2021. 146

[238] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Automated memory leak fixing on value-flow slices for C programs. In Sascha Ossowski, editor, Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016, pages 1386–1393. ACM, 2016. 27

[239] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. A survey of deep learning techniques for neural machine translation. CoRR, abs/2002.07526, 2020. 27

[240] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In 44th IEEE/ACM 44th

International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pages 1506–1518. ACM, 2022. 28, 95

[241] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. Knowledge-based environment dependency inference for python programs. In 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pages 1245–1256. ACM, 2022. 32, 73, 191, 193, 194, 203, 214

[242] Weizhao Yuan, Hoang H Nguyen, Lingxiao Jiang, Yuting Chen, Jianjun Zhao, and Haibo Yu. Api recommendation for event-driven android application development. Information and Software Technology, 107:30–47, 2019. 28

[243] Ningyu Zhang, Shumin Deng, Zhanlin Sun, Guanying Wang, Xi Chen, Wei Zhang, and Huajun Chen. Long-tail relation extraction via knowledge graph embeddings and graph convolution networks. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 3016–3025, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. 35

[244] Qirun Zhang, Wujie Zheng, and Michael R. Lyu. Flow-augmented call graph: A new foundation for taming API complexity. In Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, volume 6603 of Lecture Notes in Computer Science, pages 386–400. Springer, 2011. 28

[245] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net, 2023. 76

[246] Wujie Zheng, Qirun Zhang, and Michael R. Lyu. Cross-library API recommendation using web search engines. In SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011, pages 480–483. ACM, 2011. 28

[247] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In European Conference on Object-Oriented Programming, pages 318–343. Springer, 2009. 30

[248] Yu Zhou, Xinying Yang, Taolue Chen, Zhiqiu Huang, Xiaoxing Ma, and Harald C. Gall. Boosting API recommendation with implicit feedback. CoRR, abs/2002.01264, 2020. 28, 29

[249] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, pages 341–353. ACM, 2021. 28, 95

[250] Łukasz Langa. PEP 589 – type hinting generics in standard collections, March 2019. https://www.python.org/dev/peps/pep-0585/. 26, 34, 65, 95

[251] Radim Řehůřek and Petr Sojka. Software framework for topic modelling with large corpora. pages 45–50, 05 2010. 56