Universität Stuttgart

Supporting Software Evolution via Search and Prediction

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von
Luca Di Grazia
aus Jesi, Italien

Hauptberichter:	Prof. Dr. Michael Pradel
1. Mitberichter:	Prof. Dr. Georgios Gousios
2. Mitberichter:	Prof. Dr. Kathryn T. Stolee
Tag der mündlichen Prüfung:	1 Februar, 2024

Institut für Software Engineering der Universität Stuttgart

2024

Supporting Software Evolution via Search and Prediction
Such- und Vorhersageverfahren zur Unterstützung von Software-Evolution

Doctoral dissertation
Submitted by Luca Di Grazia
Advised by Prof. Dr. Michael Pradel

Oral defense on February 1, 2024
Examination committee:
Prof. Dr. Michael Pradel
Prof. Dr. Georgios Gousios
Prof. Dr. Kathryn T. Stolee

Institut für Software Engineering
Fakultät für Informatik, Elektrotechnik und Informationstechnik
Universität Stuttgart

Abstract

Software evolution involves the growth and adaptation of software throughout its lifecycle, including bug fixes, security patches, new programming language features, and user-driven improvements. The effective understanding and management of software evolution is essential to ensure the sustained functionality and reliability of digital systems.

In the realm of software evolution, we have identified three key challenges. The first challenge involves the necessity for improved information retrieval methods in software evolution. Efficiently identifying and localizing code changes as software evolves is a fundamental problem, for example in identifying the root cause of a bug, necessitating systematic methodologies for effective information retrieval. The second challenge underlines the importance of understanding developers' code change patterns to optimize development workflows. Analyzing commit patterns, repository characteristics, and collaborative development helps to build better methodologies and approaches for practitioners. The third challenge is that developers spend large amounts of time manually fixing bugs, and this task can be automated. This challenge is particularly crucial in large-scale projects, helping to improve development processes, enhance software maintenance, and ensure the delivery of high-quality software by automatically fixing bugs.

This dissertation argues that these challenges of software evolution can be effectively addressed through a combination of program analysis, information retrieval, and deep learning approaches. We make four contributions to support our argument. We address the first challenge with a survey on code search and proposing a search engine for code changes. For the second challenge, we conduct a comprehensive empirical study using static program analysis on the evolution of type annotations in Python to comprehend these code change patterns. For the final challenge, we introduce PyTy, an automated program repair tool for Python type errors using transfer learning.

The contributions of this dissertation impact both developers and researchers in the field. First, our survey not only enhances understanding but also guides researchers towards emerging trends and unresolved issues in information retrieval for code. DiffSearch provides developers with a powerful approach for efficiently searching code changes, demonstrating superiority over existing methods. Our empirical study on type annotations contributes valuable insights for the Python community, shedding light on adoption trends and correlations with type errors. Finally, PyTy surpasses state-of-the-art techniques, offering developers an effective and precise approach to automatically fix Python type errors. As a result, these research projects address evolving challenges and assist professionals in the important field of software evolution.

Zusammenfassung

Die Softwareentwicklung umfasst das Wachstum und die Anpassung von Software während ihres gesamten Lebenszyklus, einschließlich Fehlerkorrekturen, Sicherheitspatches, neuer Programmiersprachenfunktionen und benutzergesteuerter Verbesserungen. Ein effektives Verständnis und Management der Softwareentwicklung ist unerlässlich, um die dauerhafte Funktionalität und Zuverlässigkeit digitaler Systeme zu gewährleisten.

Auf dem Gebiet der Softwareentwicklung haben wir drei zentrale Herausforderungen identifiziert. Die erste betrifft verbesserte Methoden zur Informationsgewinnung in der Softwareentwicklung. Die effiziente Identifizierung und Lokalisierung von Code-Änderungen während der Softwareentwicklung ist ein grundlegendes Problem, z. B. bei der Identifizierung der Ursache eines Fehlers, und erfordert systematische Methoden für eine effektive Informationsbeschaffung. Die zweite Herausforderung ist das Verstehen von Code-Änderungsmustern von Entwicklern, um Entwicklungsabläufe zu optimieren. Die Analyse von Commit-Mustern, Repository-Merkmalen und kollaborative Entwicklung helfen dabei, bessere Methoden und Ansätze für Anwender zu entwickeln. Die dritte Herausforderung besteht darin, dass Entwickler viel Zeit für die manuelle Fehlerbehebung aufbringen. Diese Aufgabe kann automatisiert werden. Dies ist vor allem bei großen Projekten von entscheidender Bedeutung, da sie dazu beiträgt, die Entwicklungsprozesse zu verbessern, die Softwarewartung zu optimieren und die Bereitstellung qualitativ hochwertiger Software durch die automatische

Behebung von Fehlern zu gewährleisten.

In dieser Dissertation wird argumentiert, dass diese Herausforderungen in der Softwareentwicklung durch eine Kombination aus Programmanalyse, Information Retrieval und Deep Learning-Ansätzen effektiv angegangen werden können. Wir leisten vier Beiträge, um unser Argument zu untermauern. Wir befassen uns mit dem ersten Thema, indem wir einen Überblick über die Codesuche geben und eine Suchmaschine für Code-Änderungsmuster vorschlagen. Für das zweite Problem führen wir eine umfassende empirische Studie mit statischer Programmanalyse über die Entwicklung von Typannotationen in Python durch, um diese Codeänderungsmuster zu verstehen. Für die letzte Herausforderung stellen wir PyTy vor, ein automatisches Programmreparaturwerkzeug für Python-Typfehler unter Verwendung von Transfer Learning.

Die Beiträge dieser Dissertation wirken sich sowohl auf Entwickler als auch auf Forscher in diesem Bereich aus. Erstens verbessert unsere Studie nicht nur das Verständnis, sondern gibt den Forschern auch Hinweise auf neue Trends und ungelöste Probleme beim Information Retrieval für Code. DiffSearch bietet Entwicklern einen leistungsstarken Ansatz für die effiziente Suche nach Codeänderungen und ist damit den bestehenden Methoden überlegen. Unsere empirische Studie zu Typ-Annotationen liefert wertvolle Erkenntnisse für die Python-Community, indem sie Trends bei der Einführung und Korrelationen mit Typ-Fehlern aufzeigt. Schließlich übertrifft PyTy den Stand der Technik und bietet Entwicklern einen effektiven und präzisen Ansatz, um Python-Typfehler automatisch zu beheben. All diese Forschungsprojekte gehen auf die neuesten Herausforderungen ein und unterstützen Fachleute auf dem wichtigen Gebiet der Softwareentwicklung.

Acknowledgements

I want to say a big thank you to my supervisor, Prof. Dr. Michael Pradel. He has been a great help and guide to me. He taught me that when things seem tough, or when the idea of giving up crosses your mind, you must always remember to use the incredible inner strength everyone possess. He stimulated me to search for that crazy determination that makes you feel invincible, which gives you the certainty that everything is going to be fine. After all, maybe you will not always win, but surely you will never lose! I am also grateful to the members of my Ph.D. committee Prof. Dr. Georgios Gousios and Prof. Dr. Kathryn T. Stolee for their advice and support, their amazing work inspired me and make me grow as a researcher. A shoutout to everyone in the SOLA team and the visitors during these years, they were not only colleagues, but also friends. Many of you have been with me since the beginning, and your help has been essential. I also want to thank Katharina Plett who helped me to translate the abstract of this thesis into German and Dr. Daniel Lehmann who gave feedback on this dissertation. My wife Aubrey King deserves a special mention. Thank you for being my rock and the best part of my life. And to my family in Italy, without you all, I would not be where I am today. I cannot forget my friends, my second family, who have been there for me during the tough times. You are the best! Lastly, thanks to Prof. Dr. rer. nat. Frank Leymann, the ESE group, the PSG team at Uber, the University of Stuttgart, the state of Baden-Württemberg, the German Research Foundation (DFG), and the European Research Council (ERC). Grazie mille!

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Software Evolution

Software is always growing and evolving [168]. From the first lines of code to the eventual retirement of a program, software changes to fix bugs, patch securities, better technologies, and the needs of the people using it. This ongoing process is called software evolution. Understanding and managing software evolution is crucial, because it is essential to keep technology working properly and in a solid way.

Lehman et al. [139] are pioneers in the research field of software evolution. Their work, grounded in empirical studies of large-scale industrial systems such as IBM's OS360, led to the formulation of *Lehman's Laws of Evolution*. Key laws include the inevitability of continuing change, the tendency for increasing complexity unless explicitly addressed, the necessity of continuing growth to meet user needs, and the challenge of declining quality over time.

In addition, Godfrey et al. [70] outline a range of open questions and research limitations in the software evolution field. These include suggestions on using approaches and empirical studies to understand software evolution, and the relevance of studying open-source software.

In 2023, there were over 4.5 billion developer contributions on GitHub, and this number continues to rise annually.[1] Dealing with this massive amount of data calls for innovative solutions to analyze and understand what developers are doing and guide them, especially when it comes to code changes.

By addressing specific problems of this research field, we aim to contribute not only to better understanding software evolution but also with guidelines and approaches that can be used by developers to build resilient software during its evolution.

1.2 Challenges

We have identified three specific challenges in the field of software evolution. These challenges are:

- Challenge 1 (C-1): Better information retrieval for software evolution.
- Challenge 2 (C-2): Understanding patterns of software evolution made by developers.
- Challenge 3 (C-3): Automatically performing code changes to fix bugs.

In the rest of this section, we provide an overview of these challenges, highlighting the problems and the opportunities.

1.2.1 Better Information Retrieval for Software Evolution

One of the fundamental challenges in software evolution lies in the efficient identification and localization of code changes. As software evolves, retrieving code changes and understanding where code edits are made become a non-trivial task to learn from previous bug fixes and to build new datasets. This challenge emphasizes the need for methodologies and approaches that focus on the task of searching for code changes, ensuring a systematic and effective approach for this information retrieval problem.

[1] https://github.blog/2023-11-08-the-state-of-open-source-and-ai/

1.2.2 Understanding the Evolution Patterns of Developer Code Changes

This challenge is crucial for optimizing development workflows. Researchers need to analyze developer behavior, including commit patterns and adoption of new programming language features, to gain insights into how developers perform code changes. Exploring software evolution history, code changes and collaborative development strategies help the development of improved methodologies to ensure developers have guidelines for effective software evolution, including code maintenance.

1.2.3 Automatically Performing Code Changes to Fix Bugs

Automating code changes, particularly in the context of bug fixing, is a significant challenge in software evolution. The necessity for automated approaches that can fix bugs, without compromising the integrity of the codebase, is critical. This is particularly relevant in large-scale projects where the manual fix of every bug can be overwhelming for the developers. Moreover, automating code changes to fix bugs contributes to the overall reliability and stability of software systems, ensuring that issues are accurately addressed. The successful implementation of automated bug-fixing mechanisms not only streamlines the development process but also leads to the delivery of high-quality software.

Summary: We have identified three key challenges. The first challenge involves the necessity for improved information retrieval methods in software evolution. The second challenge underlines the importance of understanding developers' code change patterns to optimize development workflows. The third challenge is that developers spend large amounts of time manually fixing bugs, and this task can be automated.

1.3 Thesis Statement

Given the three challenges discussed above, this dissertation argues that:

Thesis Statement: The technical challenges of software evolution can be addressed using a mix of program analysis, information retrieval, and deep learning approaches.

The usage of program analysis, information retrieval, and deep learning (DL) in the context of software evolution can provide a comprehensive and effective approach to address the selected challenges by their complementary strengths:

- **Program Analysis.** We choose program analysis due to its precision in understanding code [8, 54, 199]. It can produce a detailed analysis of source code, allowing for the identification of potential bugs, optimization opportunities, and overall enhancement of code quality. By checking codebases, program analysis provides a useful and accurate understanding, contributing to effective debugging and code improvement processes [72].
- **Information Retrieval.** We motivate the selection of information retrieval by its capability to handle the vast amount of data [53, 74]. In the context of software development, especially in open-source projects with numerous contributors, managing and extracting relevant information from large datasets become crucial. Information retrieval techniques assist in efficiently navigating through these vast datasets [147], facilitating the extraction of valuable insights, trends, and key information related to code changes, repository contributions, and collaborative development efforts [130].
- **Deep Learning.** We use deep learning because it is proficient in capturing statistical patterns [71]. As software continually evolves, deep learning algorithms excel in discerning complex patterns and trends that may not be apparent through rule-based approaches. Leveraging neural networks and the attention mechanism [255], deep learning enhances the ability to recognize and adapt to software evolution, providing valuable help for potential issues and opportunities for code optimization [280].

This integrated approach not only addresses the challenges posed by software evolution but also enhances the efficiency and efficacy of the entire development lifecycle. As a result, this dissertation tackles these three challenges through research projects that focus on advancing automated code change search engines (C-1), conducting an empirical study on software evolution to better understand specific code change patterns made by developers (C-2), and developing an automatic program repair technique to empower software developers with an effective and precise approach (C-3).

1.4 Contributions

To support the thesis statement given above, we present four original research contributions. Figure 1.1 shows the overview of approaches presented in this dissertation. These works describe the state of the art of code search (Chapter 2), advance the fields of searching for code changes (Chapter 3), study the evolution of type annotations in Python (Chapter 4), and the field of automatic program repair for Python type errors (Chapter 5). In this section, we outline the contributions and research methods [241] faced in each phase of our research and detail the specific contributions made.

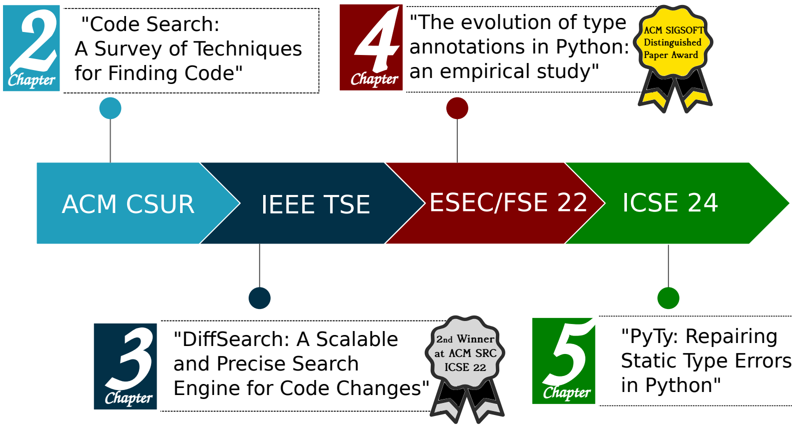

2 Chapter
"Code Search: A Survey of Techniques for Finding Code"
4 Chapter
"The evolution of type annotations in Python: an empirical study"
ACM SIGSOFT Distinguished Paper Award
ACM CSUR
IEEE TSE
ESEC/FSE 22
ICSE 24
3 Chapter
"DiffSearch: A Scalable and Precise Search Engine for Code Changes"
2nd Winner at ACM SRC ICSE 22
5 Chapter
"PyTy: Repairing Static Type Errors in Python"

Figure 1.1: Overview of the topics covered in this dissertation.

1.4.1 Code Search Survey

Chapter 2 summarizes 30 years of research on code search, giving a comprehensive overview of challenges and techniques that address them. We discuss the kinds of queries that code search engines support, how to preprocess and expand queries, different techniques for indexing and retrieving code, and ways to rank and prune search results.

Research Method: Literature Review. A comprehensive literature review serves as the foundational research method, involving the collection and synthesis of existing knowledge and research findings related to code search engines.

Contributions. Our survey reveals promising directions for future investigations. Diversifying search scenarios emerges as a key prospect, including cross-language searches and delving into version histories. Integrating code search with tools like code completion and clone detection presents an exciting frontier, and the integration of advanced deep learning models holds significant promise for improving search outcomes, potentially encouraging wider adoption by developers. Additionally, recognizing the necessity for standardized datasets and benchmarks is interesting, facilitating consistent evaluation and comparison as crucial steps in advancing the field of code search.

Challenge: C-1. To have an impact on the challenge C-1 of achieving better information retrieval for code changes, we need to understand the state-of-the-art in information retrieval for code. Navigating the extensive literature of code search engines is not an easy job. The sheer volume of existing research can be overwhelming, making it difficult to understand the current state of the art and focus on areas that could benefit from improvements. Providing a forward-looking perspective on future work, we guide researchers and developers in aligning their efforts with emerging trends and unresolved issues for C-1.

1.4.2 Searching for Code Changes

Chapter 3 presents DiffSearch, a search engine for code changes that, given a query that describes a code change, returns a set of changes that match the query in a few seconds.

Research Method: Technical Tool Development. This research method involves the development of an approach, *DiffSearch*, designed to address specific challenges in code change retrieval.

Contributions. DiffSearch is enabled by three key contributions. First, we present a query language that extends the underlying programming language with wildcards and placeholders, providing an intuitive way of formulating queries that is easy to adapt to different programming languages. Second, to ensure scalability, the approach indexes code changes in a one-time preprocessing step, mapping them into a feature space, and then performs an efficient search in the feature space for each query. Third, to guarantee precision, i.e., that any returned code change indeed matches the given query, we present a tree-based matching algorithm that checks whether a query can be expanded to a concrete code change.

Challenge: C-1. We continue to address C-1, designing a search engine that searches for code changes. Our approach supports different usage scenarios to address C-1. First, the approach supports users interested in finding *one* specific code change, e.g., when searching through the history of their own project to find some change done by a colleague. Second, DiffSearch supports users interested in finding *multiple* code changes, e.g., when searching through a set of popular open-source projects to find examples of typical ways to refactor a specific API usage. Third, the approach supports users interested in finding *many* code changes, e.g., to build a large-scale dataset to train a neural model. Finally, DiffSearch can also be configured to retrieve *all* code changes that match a query, e.g., to quantify how often specific changes occur in practice.

1.4.3 Study of Python Type Annotation Evolution

Chapter 4 presents the first large-scale empirical study on the evolution of type annotations in Python. The goal of this study is to understand the evolution and adoption rate of Python type annotations, as well as to identify any patterns or strategies that developers may follow when using them.

Research Method: Empirical Study. This research method involves an empirical study focused on analyzing real-world Python projects to investigate the adoption and evolution of type annotations.

Contributions. Our work analyzes a large dataset comprising over 9,655 Python projects and 1,123,393 commits. This deep investigation aims to address questions surrounding adoption rates, the evolution of type annotations, and their correlation with type errors. Our main contributions are the findings extracted from this comprehensive study, that offer valuable insights for both researchers and developers. For example, there is a positive trend in the adoption of type annotations, presenting an important opportunity for both researchers and practitioners to explore and leverage this evolving trend. Once introduced, type annotations tend to persist for a long time, emphasizing their lasting impact on repositories. Furthermore, the study underlines that not every element requires annotation; self-explanatory elements often do not need type annotations. Finally, the study finds that a higher volume of annotations correlates with increased efficacy in detecting type errors.

Challenge: C-2. We address C-2 by extracting a huge number of type annotations from commits, a crucial step in analyzing a large dataset focusing on these patterns in software evolution and Python type errors. The rise in popularity of type annotations suggests developers should adopt this practice, especially in projects with numerous contributors. Our study supports adding type annotations, revealing benefits such as increased type error detection with more type annotations. Improving the integration of type checking into the development process, given that many commits contain type errors but are still committed, presents a promising research direction.

1.4.4 Automatic Program Repair for Python Type Errors

Chapter 5 introduces PyTy, the first Automatic Program Repair (APR) approach designed specifically to fix static type errors in Python. Guided by a preliminary study on how developers fix type errors, our findings reveal recurring fix patterns and underscore the value of location and error message information provided by type checkers. Based on this study, PyTy is designed as a data-driven approach using a pre-trained model and transfer learning.

Research Method: Preliminary Study & Technical Tool Development. This research method combines a preliminary study and the development of a technical tool (*PyTy*) based on the findings of the preliminary study.

Contributions. We start with a preliminary study to understand strategies used by developers when fixing Python type errors. The objective is to find insights that contribute to the development of an automated approach, improving the type error fixing process for Python. After the preliminary study, we introduce PyTy, an automated program repair tool designed to address type errors in Python. This approach integrates gradual type checking with delta debugging and uses cross-lingual transfer learning. Moreover, PyTy showcases its effectiveness by surpassing previous techniques in terms of error removal. To substantiate these claims, we build and use the PyTyDefects dataset, comprising real-world type error-fix pairs sourced from GitHub repositories. Empirical evaluations demonstrate the effectiveness of PyTy in efficiently repairing type errors using benchmarks and in the wild with 20 pull requests accepted by developers.

Challenge: C-3. We address C-3, tackling the task of automatically fixing type errors in Python. We build PyTyDefects and we use it in combination with a transfer learning approach. We use a type checker as fault localization and to automatically verify the fix. Overcoming the previous limitations is crucial to establish a robust automated system that can reliably address and repair type errors in Python. These advancements to address C-3 can help developers to build more efficient software fixing type errors in Python.

Summary: We present four original research contributions: a survey on code search, a scalable and precise search engine for code changes, a large-scale study of type annotation evolution in Python, and an automated program repair for Python type errors. These contributions collectively advance the state of the art in software evolution, empowering developers with efficient and effective approaches to tackle the selected challenges.

1.5 Publications and Resources

This thesis is based on four research projects. Three of them have already been published in top-tier journals and conferences. We have used information from these past works in this thesis. Also, we have freely shared our implementation and datasets so others can use and test them (Table 1.1).[2] This helps others check our work, use it in their own studies, and makes our tools easy to obtain for anyone interested.

- Chapter 2: L. Di Grazia, M. Pradel. 'Code Search: A Survey of Techniques for Finding Code'. In: ACM Comput. Surv. (2022) [42].
- Chapter 3: L. Di Grazia, P. Bredl, M. Pradel. 'DiffSearch: A Scalable and Precise Search Engine for Code Changes'. In: IEEE Transactions on Software Engineering 49.4 (2023) [41]. **Second winner at ICSE 2022 Student Research Competition**.
- Chapter 4: L. Di Grazia, M. Pradel. 'The Evolution of Type Annotations in Python: An Empirical Study". In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2022 [43]. **Won ACM SIGSOFT Distinguished Paper Award**.
- Chapter 5: Y. Chow, L. Di Grazia, M. Pradel. "PyTy: Repairing Static Type Errors in Python". 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE) [37].

[2] https://github.com/acmsigsoft/open-science-policies

Table 1.1: Mapping between chapters and their respective resources.

Chapter	Implementation and Datasets
Chapter 2	–
Chapter 3	http://diffsearch.software-lab.org
Chapter 4	https://github.com/sola-st/PythonTypeAnnotationStudy
Chapter 5	https://github.com/sola-st/PyTy

Chapter 2

Code Search: A Survey of Techniques for Finding Code

The immense amounts of source code provide ample challenges and opportunities during software development. To handle the size of code bases, developers commonly search for code, e.g., when trying to find where a particular feature is implemented or when looking for code examples to reuse. To support developers in finding relevant code, various code search engines have been proposed. This chapter surveys 30 years of research on code search, giving a comprehensive overview of challenges and techniques that address them. We discuss the kinds of queries that code search engines support, how to preprocess and expand queries, different techniques for indexing and retrieving code, and ways to rank and prune search results. Moreover, we describe empirical studies of code search in practice. Based on the discussion of prior work, we conclude the chapter with an outline of challenges and opportunities to be addressed in the future.

2.1 Introduction

Many kinds of information are stored in digital systems, which offer convenient access, large storage capacities, and the ability to process information automatically. To enable people to quickly find digitally stored information, research on information retrieval has led to powerful search engines. Today, commercial search engines are used by billions of people every day to retrieve various kinds of information [229], such as textual information, images, or videos.

As software is becoming increasingly important in various aspects of our lives, a particular kind of information is being produced in incredibly large amounts: source code. A single, complex software project, such as the Linux kernel or modern browsers, easily comprises multiple millions of lines of source code. At the popular open-source project platform GitHub, more than 60 million new projects have been created in 2020 alone [68]. The sheer amount of existing source code leads to a situation where most code to be written by a developer either has already been written elsewhere, or at least, is similar to some code that has already been written [103, 210, 279].

To benefit from existing source code and to efficiently navigate complex code bases, software developers often search for code [223]. For example, a developer may search through a code base she is working on to find where some functionality is implemented, to understand what a particular piece of code is doing, or to find other code locations that need to be changed while fixing a bug. Beyond the code base a developer is working on, developers also commonly search through other projects within an organization or through open-source projects. For example, a developer may look for examples of how to implement a specific functionality, search for usage examples of an application programming interface (API), or simply cross-check newly written code against similar existing code. We call these and related activities *code search*. To support developers during code search, *code search engines* automatically retrieve code examples relevant to a given query from one or more code bases.

At a high level, the challenges for building a successful code search engine are similar to those in general information retrieval: provide a convenient querying interface, produce results that match the given query, and do so efficiently. Beyond these high-level similarities, code search comes with interesting additional opportunities and challenges. As programming languages have a formally defined syntax, one can unambiguously parse source code, and then analyze and compare it based on its structural properties [1]. Moreover, source code also has well-defined run-time semantics, as given by the specification of the programming language, e.g., for Java [73], C++ [104], or JavaScript [50]. That is, in contrast to natural language text and other kinds of information targeted by search engines, the meaning of a piece of source code is, at least in principle, well defined. In practice, the code in a large code corpus often is written in a diverse set of programming languages, building on various frameworks and libraries, and using different coding styles and conventions [101]. As a result, code search engines must strike a balance between precisely analyzing code in a specific language and supporting a wide range of languages [230]. Finally, the language in which a query is formulated may not be the same as the language the search results are written in. For example, many code search engines accept natural language queries or behavioral specifications of the code to retrieve, which requires some form of mapping between such queries and code [182].

Motivated by the need to search through the huge amounts of available source code and by the challenges and opportunities it implies, code search has received significant attention by researchers and practitioners. The progress made in the field is good news for developers, as they can benefit from increasingly sophisticated code search engines. At the same time, the impressive amount of existing work makes it difficult for new researchers and interested non-experts to understand the state-of-the-art and how to improve upon it. This chapter summarizes existing work on code search and describes how different approaches relate to each other. By providing a comprehensive survey of 30 years of work on code search, we hope to provide an overview of this thriving research field. Based on our discussion of existing work, we also point out open challenges and opportunities for future research.

Figure 2.1 shows the number of papers we discuss per year of publication, illustrating the increasing relevance of the topic. Our survey primarily targets full research papers, i.e., more than six pages, from top-ranked conferences and journals.[3] In addition, we include other publications, e.g., in workshop

[3] Specifically, venues ranked A* or A in the CORE ranking: http://portal.core.edu.au

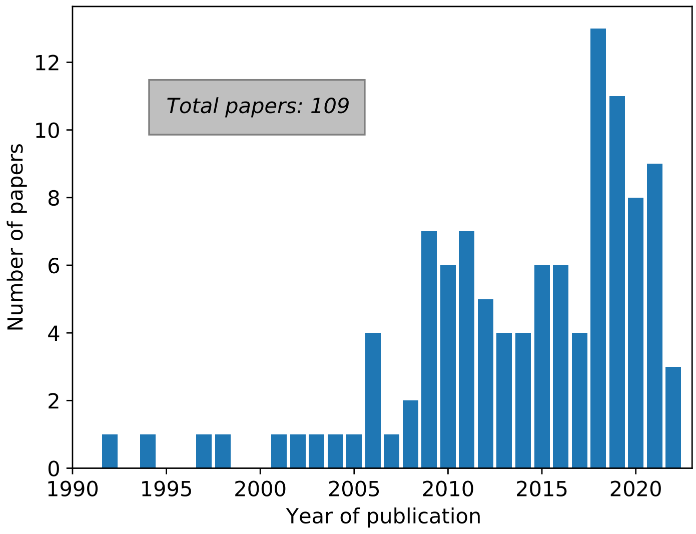


Figure 2.1: Papers on code search discussed in this chapter.

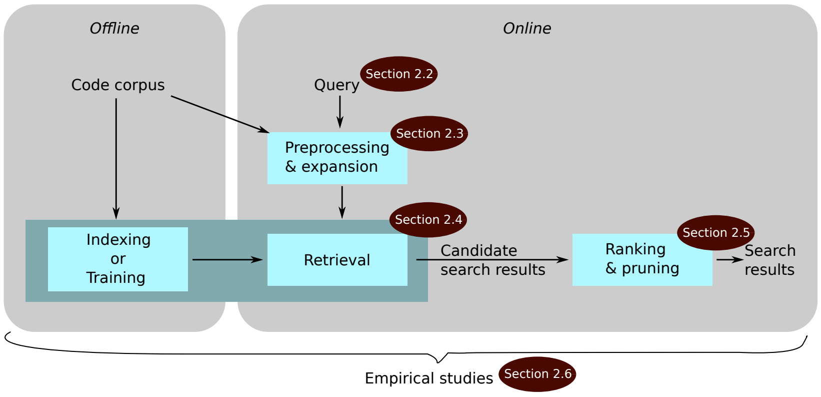


Figure 2.2: Overview of the topics covered in this chapter.

proceedings, papers on arXiv, and technical reports, as well as publications at lower-ranked venues, if and only if they are recent (less than two years), have had a significant impact (more than ten citations), or provide a very strong match with the topic of this survey. We use three different platforms to search for papers: Google Scholar[4], the ACM Digital Library[5], and DBLP[6]. To find an initial set of papers, we search with queries "code search" and "code retrieval". Afterwards, we iteratively refined the set of considered papers by following citations, both backward and forward, until reaching a fixed point.

There are several research fields related to code search that are out of the scope of this chapter. In particular, we do not discuss in detail work on general software repository mining, e.g., to extract patterns or programming rules [113], searching for entire applications, e.g., in an app store [77, 164], and query-based synthesis of new code examples [84]. Moreover, we do not cover in detail work on code clone detection [221] and code completion [25, 214], as those are related but different problems. Code clone detection aims at finding pieces of code that are semantically, and perhaps even syntactically, equivalent to each other, whereas code search aims at finding code that offers more details than a given query. Code completion can be seen as a restricted variant of code search, where the code a developer has already written serves as a query to find the next few tokens or even lines to insert. An important difference is that code search tries to retrieve existing code as-is, whereas code completion synthesizes potentially new code fragments.

Figure 2.2 outlines the components a typical code search engine is built from, and at the same time, gives an overview of the topics covered in this chapter. Most code search engines have an offline part, which indexes a code corpus or trains a machine learning model on a code corpus, and an online part, which takes a user-provided query and retrieves code examples that match the query.

- Section 2.2 presents different kinds of queries accepted by code search engines, including natural language, code snippets, formal specifications, test cases, and queries written in specifically designed querying languages.

[4]https://scholar.google.com/
[5]https://dl.acm.org/
[6]dblp.uni-trier.de/

- Section 2.3 describes how code search engines preprocess and expand a given query, e.g., by generalizing terms in a natural language query or by lifting a given code snippet to a richer representation.
- Section 2.4 presents the core component of a code search engine, which indexes code examples or trains a machine learning model, and then retrieves examples that match a query. We discuss and compare several approaches based on how they represent the code and what kind of retrieval technique they use.
- Section 2.5 presents different techniques for ranking and pruning search results before presenting them to the user, e.g., based on similarity scores between code examples and the query, or based on clustering similar search results.
- Section 2.6 discusses empirical studies of developers and how they interact with code search engines, which connects the research described in the other sections to adoption in practice.
- Section 2.7 outlines several open challenges and research directions for future work.

Prior work surveys code search techniques from different perspectives than this chapter. Garcia et al. [66] summarize code search-related tools presented until 2006, with a focus on tools aimed at software reuse. Another survey [46] is about techniques for locating where in a project a particular feature or functionality is implemented. While being a problem related to code search, feature location focuses on searching through a single software project, instead of large code corpora, and on the specific use case of locating a feature, instead of the wider range of use cases covered by code search. A short paper by Khalifa [120] discusses existing techniques for code search, focusing on information retrieval-based and deep learning-based approaches, but it covers only five papers. Finally, another survey of code search techniques [147] focuses on general publication trends, application scenarios where code search is used, and how search engines are evaluated. In contrast, this chapter focuses more on the technical core of code search engines, including different querying languages, pre-processing of queries, ranking and pruning of results, and also empirical studies of code search in practice.

2.2 Queries for Searching Code

The starting point of every search is a query. We define a query as an explicit expression of the intent of the user of a code search engine. This intent can be expressed in various ways, and different code search engines support different kinds of queries. The designers of a code search engine typically aim at several goal when deciding what kinds of queries to support:

- *Ease*. A query should be easy to formulate, enabling users to use the code search engine without extensive training. If formulating an effective query is too difficult, users may get discouraged from using the code search engine.
- *Expressiveness*. Users should be able to formulate whatever intent they have when searching for code. If a user is unable to express a particular intent, the search engine cannot find the desired results.
- *Precision*. The queries should allow specifying the intent as unambiguously as possible. If the queries are imprecise, the search is likely to yield irrelevant results.

These goals are non-trivial to reconcile, and different code search techniques balance this trade-off in different ways. Figure 2.3 shows a taxonomy of the kinds of queries supported by existing approaches. Broadly, we can distinguish between informal queries, formal queries, and combinations of the two. The numbers associated with the leaf nodes of the taxonomy indicate how many papers support each kind of query. The figure also shows how well different approaches achieve the three goals from above. The color of the boxes containing "Ease", "Precision", and "Expressiveness" indicate the support for these goals, where green means strong support, yellow means medium support, and red means little support. The remainder of this section discussed the different kinds of queries in more detail, following the structure lined out in the taxonomy.

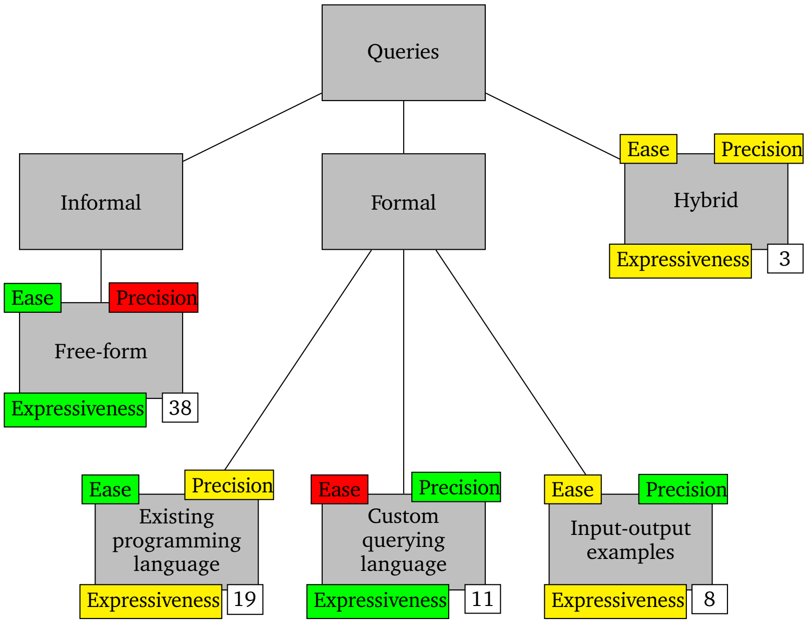


Figure 2.3: Taxonomy of code search queries and number of approaches accepting each kind of query.

2.2.1 Free-Form Queries

Free-form queries are an informal way of specifying the intent of a code search. Such a query may describe in natural language the functionality of the searched code, e.g., "read file line by line". Free-form queries may also contain programming language elements, e.g., when searching for identifier names of a specific API, such as "FileReader close".

Free-form queries are the most commonly used kind of query in the approaches we survey [2, 12, 29, 30, 31, 34, 44, 48, 49, 80, 82, 95, 96, 143, 145, 146, 148, 149, 151, 152, 156, 159, 165, 169, 182, 184, 208, 209, 222, 227, 230, 236, 246, 256, 260, 261, 268, 276, 282]. They are attractive as users can easily formulate a query, similar to using a general-purpose web search engine, with a high level of expressiveness. On the downside, free-form queries risk being imprecise. One reason is that natural language terms are ambiguous. For example, the term "float" may refer to either a data type or to a verb. Another reason is that the vocabulary in a query may not match the vocabulary used in a code base. For example, a search term "array" may refer to a data structure that syntactically occurs as two square brackets in Java or Python [256].

Because free-form queries are extremely versatile, different code search engines compare them against different kinds of data. One set of approaches compares free-form queries against natural language text associated with code, e.g., API documentation [30], commit messages [31], or words in the a project's metadata [165]. Another set of approaches compares queries against the source code, e.g., by matching the query against signatures of fields and methods [95, 96] or against all identifiers in the code [152, 159, 230].

The informal nature of free-form queries may make it difficult to accurately match a query against a code snippet, e.g., because of a vocabulary mismatch between the two. For example, plain English queries, such as "match regular expression" or "read text file" [208], may not match the terms used in the corresponding API methods. A popular way to mitigate this mismatch is to project natural language words and source code identifiers into a common vector space [182] via learned word embeddings, such as Word2Vec [170]. Another way to address the limitations of free-form queries is to preprocess and expand queries, which we discuss further in Section 2.3.

To avoid the ambiguity of free-form queries and because source code is anyway written in a formal language, many code search engines support some kind of formal queries, which we discuss in the following. The commonality of these queries is that they are written in a language with a formally specified syntax, and sometimes also formally defined semantics.

Summary: Free-form queries are easy to type and highly expressive, but they can be ambiguous and less precise than other, more formal kinds of queries.

2.2.2 Queries Based on Existing Programming Languages

As a first kind of formal query, we start by discussing queries based on existing programming languages. A query here is a snippet of code, possibly using some additional syntax not available in the underlying programming language. Because developers already know the programming language they are using, such queries are easy to formulate. The expressiveness and precision of code queries varies depending on the intent of the user and the specific search engine.

Queries based on existing programming languages roughly fall into three categories:

1. *Plain code*. The most simple kind of code query are snippets of code as defined by the syntax of the underlying programming language [15, 62, 124, 137, 138, 154, 163, 177, 249, 286, 287]. For example, the following query provides a partial implementation, for which the user seeks ways to extend it [15]:

```
try {
  File file = File.createTempFile("foo", "bar");
} catch (IOException e) { }
```

2. *Code with holes*. Instead of letting the search engine figure out where to extend a given code snippet, some search engines support queries that explicitly define one or more holes in the given code [172, 175]. For example, this query specifies that the user looks for how to complete the body of the given method [175]:

```
public void actionClose(JButton a, JFrame f) {
  __CODE_SEARCH__;
}
```

3. *Code with pattern matching symbols*. A very precise way of describing the code to search is a query in an extension of the underlying programming language that adds patterns matching symbols. For example, such queries may define where an expression, here denoted with `#`, or a statement, here denoted with `@`, is missing [195, 196]:

```
if (# = #) @;
```

> Such a query provides an abstract template for the code to search, and the search engine tries to retrieve some or all code snippets that the template can be extended into.

A recurring challenge for search engines that accept queries written in (variants of) existing programming languages is the problem of parsing incomplete code snippets [124, 172, 266]. An off-the-shelf grammar of the programming language may not be able to parse a query because the query does not encompass a complete source code file or because the code is incomplete. One way to address this problem [124] is to heuristically fix a given code fragment, e.g., by surrounding it with additional code.

A popular kind of application of search engines that accept partial code snippets is as a source code recommendation tool. To ensure that the recommended code matches the current context a developer is working in, e.g., the current file and project, some approaches consider the code around the actual query as context available to the search engine. For example, Holmes and Murphy [99] and Brandt et al. [22] propose to integrate code search directly into the IDE. Other approaches [175, 249, 287] spontaneously search and display example code snippets while the developer is editing a program. The underlying idea of these approaches is that the user should not spend time on formulating the query, but simply uses the already typed code. Finally, general code completion systems also predict code based on the existing code context while a developer is writing code. For example, GitHub's Copilot[7] suggests multiple lines of code using a large-scale generative neural language model [33]. In contrast to code search, code completion synthesizes suitable code, regardless of whether exactly this code has already been written anywhere, whereas code search retrieves existing code as-is.

Instead of queries in a high-level programming language, some code search engines accept binary code as a query. For example, an approach by David and Yahav [40] accepts a function in its compiled, binary form as a query and then searches for similar functions in a corpus of binaries. Another approach accepts an entire binary as the query, trying to find other binaries that may be compiled from the same or similar source code [122]. Binary-level code search has several

[7] `https://copilot.github.com/`

applications in security, e.g., to check for occurrences of known vulnerable code, and in copyright enforcement, e.g., to find code copied without permission.

Summary: Program language queries are easy to type because users do not have to learn a new language, but the expressiveness and precision of code queries varies depending on the intent of the user and the specific search engine.

2.2.3 Custom Querying Languages

A common alternative to queries based on an existing programming programming language are custom querying languages. They provide a high degree of expressiveness and precision, at the expense of reduced ease of use, because users need to learn the querying language.

Logic-based Querying Languages

The most prevalent kind of custom querying languages is first-order logic predicates that describe properties of the code to search. For example, Janzen and Volder [105] extend the logic programming language TyRuBa[8] to support queries such as the following, which searches for a package with a class called "HelloWorld"

```
package(?P, class, ?C), class(?C, name, HelloWorld)
```

In a similar vein, Hajiyev, Verbaere, and de Moor [86] describe a code querying technique based on Datalog queries. Datalog is a logic-based language that, given a set of elements and relationships between the elements, answers queries. The approach considers program elements, e.g., classes and methods, and several relationships between them, e.g., the fact that a class inherits from another class, or that a class has a method. A user can query a code base by formulating logic-based queries over these elements and relationships, such as asking for all methods in a class called "A", where "A" inherits from a class called "B". Other

[8]http://tyruba.sourceforge.net/

approaches support logical queries over identifiers and structural relationships between them [234, 259, 262].

Several languages allow for predicates beyond describing program elements and their relationships. One example is to also support meta-level properties, such as how many imports a file has. For example, the query language by Martie, LaToza, and van der Hoek [161] allows for queries such as:

```
import count > 5 AND extends class FooBar
```

The Alice search engine [238] supports a kind of semantic predicates, e.g., to search for code that calls the `readNextFile` method in a loop and handles an exception of a type `FileNotFoundException`.

Significant Extensions of Existing Programming Languages

Instead of logic-based querying languages, several search engines accept queries in custom languages that significantly extend an existing programming language. Similar to the kinds of queries discussed in Section 2.2.2, such queries contain fragments of an existing programming language. One such language is by Inoue et al. [103] who support different kinds of wildcard tokens that match any single token, any token sequence, or any token sequence discarding paired brackets, respectively. In addition, their queries may use popular regular expression operators for choice, repetition, and grouping to enhance the expressiveness. For example, the following query will search for nested if-else clauses:

```
$( if $$ else $) $+
```

Another significant extension of an existing programming language is the "semantic patch language" of Coccinelle [130]. It allows for describing a patch, as produced by the popular *diff* tool, augmented with metavariables that match a specific piece of code and with a wildcard operator. A query hence describes a set of rules that the old and the new code must match, which then used to search for specific code changes in the version history of a project [129].

Other Custom Languages

A custom querying language by Premtoon, Koppel, and Solar-Lezama [205] describes code in a way that can be mapped to a program expression graph, which describes computations via operator nodes and dataflow edges [251]. In contrast to the above approaches, their queries are not specific to a single programming language, but can be used to search through projects in multiple languages.

Summary: Custom querying language queries can offer high expressiveness and precision, but are (at least initially) less easy to type because users have to learn the custom language first.

2.2.4 Input-Output Examples as Queries

All kinds of queries discussed so far focus on the source code itself, but neglect an important property of code that distinguishes it from other kinds of data supported by search engines, such as text: executability. To exploit this property, some search engines support queries that are behavioral specifications and that characterize examples of the code behavior. Such queries typically come in the form of one or more input-output examples of the code to search.

The pioneering work by Podgurski and Pierce [200] is the first to propose input-output examples as queries, and other approaches adopt and improve this idea [109, 215, 242, 243, 244]. For example, these search engines enable users to search for code that given the input “susie@mail.com” produces “susie” [242]. Beyond supporting developers who search for specific kinds of code, another application of input-output-based code search is to find code fragments that can be used in automated program repair [117].

An extended form of input-output examples are queries in the form of executable test cases [140, 141]. Adapting the test-driven development paradigm, the basic idea is that a developer first implements test cases for some functionality and then searches for existing code that provides the desired functionality. Test cases here serve two purposes: First, they define the behavior of the desired code to be searched. Second, they test the search results for suitability in the local context.

Summary: Using input-output examples as queries allows for precisely specifying the desired behavior, but providing sufficiently many examples to fully express this behavior may require some effort.

2.2.5 Hybrids of Informal and Formal Queries

A few approaches support not only one kind of query, but hybrid queries that combine multiple of the kinds described above. One example is the work by Reiss [215], which in addition to input-output examples supports free-form queries. For example, a user may search for a method that mentions "roman numeral" and produces "XVII" for the input "17". Another kind of hybrid query combines free-form, natural language terms with references to program elements [266], e.g., "sort playerScores in ascending order", where "playerScores" refers to a variable in the code. Finally, Martie, Hoek, and Kwak [160] mix free-form queries and logical queries over code properties. For example, a query may ask for code that matches the keywords "http servlet", extends a class `httpservlet`, and has more than three imports.

2.3 Preprocessing and Expansion of Queries

The query provided by a user may not be the best possible query to obtain the results a user expects. One reason is that natural language queries suffer from the inherent imprecision of natural language. Another reason is that the vocabulary used in a query may not match the vocabulary used in a potential search result. For example, a query about "container" is syntactically different from "collection", but both refer to similar concepts. Finally, a user may initially be unsure what exactly she wants to find, which can cause the initial query to be incomplete.

To address the limitations of user-provided queries, approaches for preprocessing and expanding queries have been developed. We discuss these approaches by focusing on three dimensions: (i) the user interface, i.e., if and how a user gets involved in modifying queries, (ii) the information used to modify queries, i.e., what additional source of knowledge an approach consults, and (iii) the actual technique used to modify queries. Table 2.1 summarizes different approaches along these three dimensions, and we discuss them in detail in the following.

Table 2.1: Overview of approaches for preprocessing and expansion of queries.

	Paul and Prakash [196]	Wang, Lo, and Jiang [259]	Shepherd et al. [230]	Sisman and Kak [237]	Lv et al. [156]	Lu et al. [152]	Martie, LaToza, and van der Hoek [161]	Li et al. [143]	Nie et al. [184]	Martie, Hoek, and Kwak [160]	Sirres et al. [236]	Rahman and Roy [209]	Lu et al. [151]	Wu and Yang [268]	Li et al. [142]
User interface:															
Transparent to user	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓		✓	✓
Based on (implicit) user feedback				✓			✓			✓					✓
Information used to modify queries:															
Initial search results				✓			✓			✓				✓	
Similarity of search terms				✓	✓	✓		✓	✓		✓	✓	✓		
NL/code dataset					✓				✓		✓	✓			
Recurring code changes														✓	
Technique used to modify queries:															
Weigh search terms			✓									✓			
Add or replace search terms				✓	✓		✓	✓	✓		✓	✓	✓	✓	
Lift query to richer representation	✓	✓					✓								

2.3.1 User Interface of Query Preprocessing and Expansion Approaches

Transparent vs. interactive. The majority of code search engines that perform some form of query preprocessing or expansion do so in a fully transparent way, i.e., the user is not aware of this part of the approach. For example, Sisman and Kak [237] propose to automatically expand queries using similar terms from search results, while others transparently expand queries using dictionaries [143] and synonymous [152]. An exception is the work by Martie, Hoek, and Kwak [160] and Martie, LaToza, and van der Hoek [161], where the user interactively reformulates queries based on keyword recommendations made by the search engine. Another interactive approach [151] collects relations between words from the source code, such as that one type name inherits from another, or that a word is part of a compound word, and then removes irrelevant words using an English dictionary. After this process the user can give a feedback about the query and iterate the query refinement until satisfied.

User feedback. To improve the initially given queries, some approaches rely on feedback by the user. Such feedback can be given explicitly, as in the work by Martie, Hoek, and Kwak [160] and Martie, LaToza, and van der Hoek [161], where a user can up-rate or down-rate particular search results, which is then used to show more or less search results with similar features. Instead of explicit feedback, Sisman and Kak [237] rely on so-called pseudo relevance feedback given implicitly through the highest-ranked search results retrieved for initially given query. The approach then enriches the initial query with search terms drawn from the initial search results. Another way of using implicit user feedback is by observing what search results a user clicks on, which may provide valuable information on what the user is searching for. The Cosoch approach exploits this feedback in a reinforcement learning-based approach [142]. Their approach tracks across multiple queries which search results a user selects, and tries to maximize the normalized discounted cumulative gain (NDCG), which measures the quality of a ranked list of search results.

Summary: User interfaces can help preprocessing queries in a transparent way or using feedback from users.

2.3.2 Information Used to Modify Queries

Initial search results. Effectively modifying a query requires some information in addition to the query itself. Several approaches use the results returned for the initially provided query for this purpose [160, 161, 237]. A downside of relying on initial search results to modify queries is that the search must be performed multiple times before obtaining the final search results, which may negatively affect efficiency.

Similarity of search terms and/or identifiers. A commonality of several approaches for query preprocessing and expansion is to compare words or identifiers in a query with those in a potential search result through some kind of similarity measure. One approach [237] builds on the observation that terms in search results that frequently appear close to terms in the query may also be relevant, and then expands the initial query with those words. Others builds on domain-specific dictionaries [143] or on synonyms [152] found using WordNet [135] to add or replace query terms with related terms. A more recent alternative to curated databases of word similarities are learned word embeddings, e.g., via Word2vec [170], which can help in revising queries [209].

NL/code datasets. A third kind of information used by several approaches to revise queries are datasets of documents that combine natural language and code. For example, Lv et al. [156] use API documentation to identify which API a query is likely to refer to, and then expand the query accordingly. Online discussion forums with programming-related questions and answers, e.g., Stack Overflow also have been found to help in revising queries [184, 209, 236]. These approaches search online posts related to a given query, and then extract additional relevant words, software-specific terms, and API identifiers to augment the query. Since the questions and answers cover various application domains and are curated based on feedback by thousands of developers, they provide a valuable dataset to associate natural language words with related programming terms.

Recurring code changes. Motivated by the observation that developers may have to adapt a retrieved code example, e.g., to use the most recent version of an API, Wu and Yang [268] expand queries to proactively consider such potential code adaptations. At first, their approach mines recurring code changes from version histories of open-source projects, which provides information such as that a code token `A` is often changed to a code token `B`. Given a user query, they then retrieve matching code examples, and if these examples include a frequently changed token, say `A`, expand the query with the updated token, say `B`. With the expanded query, the search engine hence will retrieve updated versions of the code examples, freeing the developer from adapting the code manually.

Summary: To automatically modify queries, code search engines most commonly use similarities between search terms and identifiers, as well as corpora of natural language and code.

2.3.3 Techniques Used to Modify Queries

Weigh search terms. The perhaps most straightforward way of augmenting a given search query is to weigh the given search terms. Several code search engines implement this idea [209, 230], with the goal of giving terms that are most relevant for finding suitable results at a higher weight. For example, Rahman and Roy [209] estimate the weights of API class names by applying the page rank algorithm [23] to an API co-occurrence graph.

Add or replace search terms. Another common technique is to add or replace terms in the given search query, e.g., by adding terms that are related or synonymous to those already in the query. Lv et al. [156] propose a query refinement technique specifically aimed at APIs. In a two-step approach, they first identify an API that the query is likely to refer to, and then expand the original query with identifier names related to this API. For example, for an initial query "how to save an image in png format", the approach may identify the API method `System.Drawing.Image.Save` to be likely relevant, and hence, adds the fully qualified method name into the search query.

Lift query to richer representation. To ease matching a query against potential search results, several approaches lift the query into a richer representation. An early example is the SCRUPLE tool by Paul and Prakash [196]. It transforms the query specified by the user with a pattern parser into an extended nondeterministic finite automaton called code pattern automaton. Other approaches lift queries into a graph representation. For example, Wang, Lo, and Jiang [259] take a query formulated in a custom querying language and then transforms it into a graph representation that expresses call relations, control flow relations, and data flow relations. As another example, Li et al. [143] transform a natural language query into an “action relationship graph”, which expresses sequencing, condition, and callback relationships between parts of the code described in the query. For example, given a query “add class ’checked’ to element and fade in the element”, their approach would infer that the two parts combined by “and” are supposed to happen in sequence.

Summary: The most popular techniques to modify queries are using weighing, adding, or replacing search terms, as well as lifting queries to a richer representation.

2.4 Indexing or Training, Followed by Retrieval of Code

The perhaps most important component of a code search engine is about retrieving code examples relevant for a given query. The vast majority of approaches follows a two-step approach inspired by general information retrieval: At first, they either index the data to search through, e.g., by representing features of code examples in a numerical vector, or train a model that learns representations of the data to search through. Then, they retrieve relevant data items based on the pre-computed index or the trained model. To simplify the presentation, we refer to the first phase as “indexing” and mean both indexing in the sense of information retrieval and training a model on the data to search through.

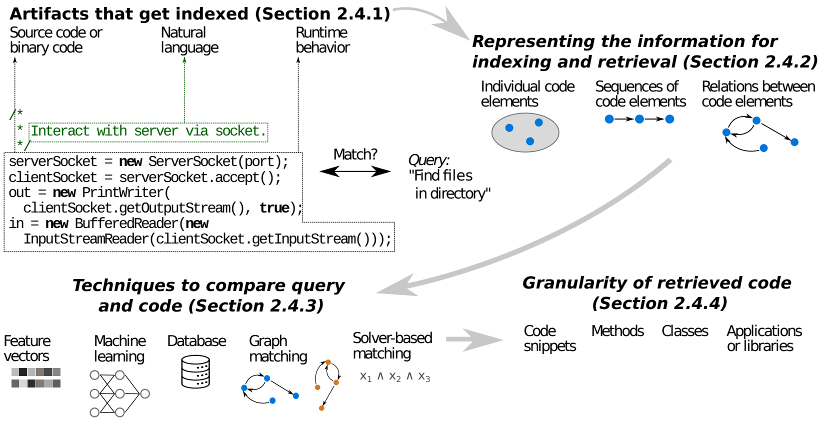
The primary goal of indexing and retrieval is effectiveness, i.e., the ability to find the “right” code examples for a query. To effectively identify these code examples, various ways of representing code and queries to compare them with each other have been proposed. A secondary goal, which is often at odds with achieving effectiveness, is efficiency. As users typically expect code search engines to respond within seconds [223], building an index that is fast to query is crucial. Moreover, as the code corpora to search through are continuously increasing in size, the scalability of both indexing and retrieval is important as well [9].

We survey the many different approaches to indexing, training and retrieval in code search engines along four dimensions, as illustrated in Figure 2.4. Section 2.4.1 discuss what kind of artifacts a search engine indexes. Section 2.4.2

Table 2.2: Overview of approaches based on information retrieval technique respect to kind of indexed information (1-3 rows) and kind of feature extracted (4-6) rows.

	IR technique		
	Feature Vectors	Machine Learning	Other
Indexed artifact:			
Source code	[14, 15, 44, 62, 122, 137, 138, 154, 182, 184, 230, 236, 268, 287]	[101, 227, 237, 238]	[40, 86, 99, 103, 143, 161, 195, 196]
Runtime behavior	[200]	[109, 172, 201, 244]	
Natural language	[12, 28, 34, 124, 165, 249]	[29, 82, 146, 175, 222, 246]	[31, 189]
Representation of indexed code:			
Individual code elements	[44, 122]	[29, 34, 227]	[31, 103]
Sequences of code elements	[12, 124, 249, 268]	[82]	[40]
Relationships between code elements	[15, 137, 138, 154, 156, 184]	[146, 172, 177, 246]	[18, 86, 99, 143, 195, 196]

describes different ways of representing the extracted information. Section 2.4.3 presents techniques for comparing queries and code examples with each other. Table 2.2 summarizes the approaches along these first three dimensions. Finally, Section 2.4.4 discusses different levels of granularity of the source code retrieved by search engines.


Artifacts that get indexed (Section 2.4.1)
Source code or binary code
Natural language
Runtime behavior
Match?
Query: "Find files in directory"
Representing the information for indexing and retrieval (Section 2.4.2)
Individual code elements
Sequences of code elements
Relations between code elements
Techniques to compare query and code (Section 2.4.3)
Feature vectors
Machine learning
Database
Graph matching
Solver-based matching
$x_1 \wedge x_2 \wedge x_3$
Granularity of retrieved code (Section 2.4.4)
Code snippets
Methods
Classes
Applications or libraries

```
/*
 * Interact with server via socket.
 */
serverSocket = new ServerSocket(port);
clientSocket = serverSocket.accept();
out = new PrintWriter(
  clientSocket.getOutputStream(), true);
in = new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));
```

Figure 2.4: Overview of techniques for indexing and retrieval.

2.4.1 Artifacts That Get Indexed

When creating an index of code examples to retrieve, code search engines consider different artifacts related to code.

Source Code and Binary Code

The most obvious, and by far most prevalent, artifact to index is the source code itself. Many approaches target a high level programming language, such as Java [11, 15, 44, 62, 86, 99, 138, 182, 238, 276, 287], JavaScript [137, 143], and C [195, 196]. Some search engines support not only one but multiple languages [29, 101, 227], e.g., Sando [230] (C, C++, and C#), ccgrep [103] (C, C++, Java, and Python), Aroma [154] (Hack, Java, JavaScript, and Python), DGMS [145] (Java and Python), and COSAL [163] (Java and Python). Instead of source code, some approaches focus on compiled code [40, 122], which is useful, e.g., to find functions in binaries that are similar to known vulnerable functions. These approaches first disassemble a given binary and then index disassembled functions or entire binaries.

Runtime Behavior of Code

Instead of only statically analyzing and indexing code, some search engines exploit the fact that source code can be executed by analyzing the runtime behavior of the code to search through. Considering runtime behavior may be useful, e.g., when two snippets of code have similar source code but nevertheless perform different behavior. The first code search engine that considers runtime behavior is by Podgurski and Pierce [200]. Their approach expects the user to provide inputs to the code to find, and then searches for suitable code examples by sampling the behavior of candidate code examples. Reiss [215] select candidates using keywords and then they apply different kinds of transformations to have solutions with different behavior. To validate the dynamic behavior of the candidates with respect to the requirements given by the user, they run a set of test suites. Another line of work symbolically executes code to gather constraints that summarize the runtime behavior [109, 244]. Finally, the COSAL search engine [163] compares the behavior of code snippets based on an existing technique for clustering code based on its input-output behavior [162].

Natural Language Information Associated with Code

Beyond the source code and its runtime behavior, another valuable artifact is natural language information associated with code. For example, such information comes in the form of comments, API documentation, commit messages, and discussions in online question-answer forums. Several code search engines leverage this information, in particular approaches that retrieve code based on natural language queries, e.g., after training neural models on pairs of natural language descriptions and source code [29, 246].

One group of approaches leverages regular comments and structured comments that provide API documentation, e.g., by considering comments as keywords to compare a query against [146] or by mapping code and natural language into a joint vector space [82, 222]. Another direction is to consider commit messages in a version control system, based on the assumption that the words in a commit message describe the source code lines affected by the commit [31].

Finally, online discussion forums, such as Stack Overflow,[9] provide a dataset of pairs of code snippets and natural language descriptions, which several code search engines use to associate natural language words and code [28, 34, 184].

Summary: The by far most common kind of artifact that gets indexed are source code and binary code. However, there also are search engines that index traces of runtime behavior and natural language information associated with code.

2.4.2 Representing the Information for Indexing and Retrieval

After discussing what artifacts different approaches extract information from, we now consider how this information is represented for indexing and retrieval. We identify three groups of approaches, presented in the following with increasing levels of complexity: representations based on individual code elements, on sequences of code elements, and on relations between code elements.

Individual Code Elements

The first group of approaches focuses on individual code elements, e.g., tokens or function calls, ignoring their order and any other kind of relationship they may be in [31, 103]. To index the code examples, these approaches then represent a code snippet as a set of code elements. One example is work that represents a code example as a bag of tokens, and a natural language query as a bag of words [34, 249]. Another approach represents binaries as a set of tokens extracted from disassembled binaries [122]. Finally, Diamantopoulos, Karagiannopoulos, and Symeonidis [44] represent API usages as a set of API calls, replacing each method by its type signature. The main benefit of indexing sets of individual code elements is the conceptual simplicity of the approach, which facilitates instantiating an idea for a particular target language. On the downside, the order of code elements and other kinds of relationships may provide useful information for precisely matching a code example against a query.

[9] https://stackoverflow.com/

Sequences of Code Elements

To preserve ordering information of code elements during the indexing, several approaches extract sequences of code elements from a given code example. Most commonly, these sequences are extracted in an Abstract Syntax Tree (AST) based, static analysis that focuses on particular kinds of nodes. For example, Gu, Zhang, and Kim [82] represent API usages by extracting sequences of API calls from an AST. Another example is FaCoY [124], which represents a code example as a sequence of tokens extracted from an AST, where each token comes with a token type, e.g., method call or string literal. To represent incomplete code examples, they insert empty statements to complete the snippet. David and Yahav [40] instead use control flow graphs to represent information for the binary source code. Sun et al. [246] represents code as a sequence of low-level instructions, which the approach obtains by compiling and then disassembling the code. Finally, deep learning-based code search approaches often tokenize source code using a sub-word tokenizer, such as the WordPiece [271] tokenizer used, e.g., by Salza et al. [227].

Relations between Code Elements

Going beyond individual code elements and sequences thereof, many approaches extract a richer set of relations between code elements. The most common approach is to focus on entities, typically code elements, such as classes, methods, and statements, and relations between them, such as one class inheriting from another class, one method calling another method, and a method containing a statement [15]. Popular examples of this approach include CodeQuest [86] and Sourcerer [146], which extract code elements and their relations through an AST-based analysis. Sourcerer also serves as the basis for other code search approaches, e.g., by Bajracharya, Ossher, and Lopes [12] and Lv et al. [156].

Sirres et al. [236] extract structural code entities of Java source files, collecting the relationship of imports, classes, methods and variables. A more recent example is Aroma [154], which parses code into a simplified parse tree and then extracts different kinds of features based on the tokens in the code, parent-child relationships, sibling relationships, and variable usages, focusing. In a similar vein, Ling et al. [145] represent code as a graph that includes structural parent-child relationships, next-token relationships, and definition-use information. Li et al. [143] extract from ASTs three kinds of relationships between code elements: sequencing methods calls, callback between methods and methods as conditions of if statements. Holmes and Murphy [99] use heuristics to collect relationships of methods inheritance, methods calls and methods usage. Finally, Paul et al. [195, 196] use non-deterministic finite automata, called code pattern automata, to represent relationships between code elements.

Instead of a relatively lightweight static extraction of information to index, some search engines rely on more sophisticated static analysis. For example, Mishne, Shoham, and Yahav [172] propose a static type state analysis that extracts temporal specifications in the form of deterministic finite-state automata that capture sequences of API method calls. Another example is work by Premtoon, Koppel, and Solar-Lezama [205], which represent code examples as data flow graphs.

Summary: To index source code examples, code search engines typically represent the code as sets of individual code elements, sequences of code elements, or as relationships between code elements.

2.4.3 Techniques to Compare Queries and Code

After extracting the information from source code, execution behavior, and natural language information associated with the code, most search engines index the extracted information to then quickly respond to queries based on the pre-computed index. The following discusses different approaches for comparing queries and code, which we group into techniques based on feature vectors computed without machine learning (Section 2.4.3), machine learning-based techniques (Section 2.4.3), database-based techniques (Section 2.4.3), graph-based matching (Section 2.4.3), and solver-based matching (Section 2.4.3).

Indexing and Retrieval Based on Algorithmically Extracted Feature Vectors

Several techniques are based on feature vectors and distances between these vectors. In this sub-section we discuss approaches that compute feature vectors algorithmically, i.e., without any machine learning model. Their general idea is to represent both the code examples and the query as feature vectors, and to then retrieve code examples with a vector similar to that of the query. Because performing a pairwise comparison of the query vector with each code vector is inefficient, the approaches compute an index into the feature space that allows them to efficiently retrieve a ranked list of vectors similar to a given vector.

There are different ways of mapping information about code examples and queries into feature vectors. One approach is boolean vectors [226] that express whether some feature, e.g., a particular type of AST node, are present [154]. Another common approach is to map a set of tokens or words into a term frequency-inverse document frequency (TF-IDF) vector, which expresses not only whether a feature is present, but also how important its presence is in comparison with other features [44, 237, 249, 268].

A popular implementation of feature-based indexing and retrieval is the Lucene library.[10] Originally designed for text search, Lucene is used in various code search engines [12, 14, 122, 124, 161, 201, 236]. It combines the boolean model, which removes candidate vectors that do not provide the required features, and the vector space model, which computes a distance between the remaining candidate vectors and the query vector. The feature vectors are based on a custom term-frequency formula.[11] Moreover, Nguyen et al. [182] use a revised Revised Vector Space Model (rVSM). The rVSM splits each token in separate words and computes the weight for each word using TF-IDF.

Instead of building upon an existing indexing and retrieval component, some search engines implement their own indexing and retrieval technique. For example, Lee et al. [138] use R*trees [18], which recursively partition the code examples into a tree structure that can then be used to efficiently find the nearest neighbors of a query. Luan et al. [154] identify those code examples that have the most overlap with the query vector by representing the feature set as a sparse vector and by then computing the overlap between queries and code examples via matrix multiplication. Another approach [15] matches a code query against code examples based on feature vectors for different AST subtrees of the code examples, pruning the large number of combinations to compare by considering

[10] https://lucene.apache.org/
[11] https://lucene.apache.org/core/3_5_0/scoring.html

only subtrees with the same parent node type.

Learning-based Retrieval

Neural software analysis [202] is becoming increasingly popular, and neural information retrieval [173] offers an attractive alternative to more traditional techniques. Most work takes an end-to-end neural learning approach, where a model learns to embed both queries and code examples into a joint vector space. Given this embedding, code search reduces to finding those code examples that are the nearest neighbors of a given query. We discuss approaches following this overall pattern in the following, focusing at first on natural language-to-code search and then on code-to-code search.

Learning-based natural language-to-code search. Gu, Zhang, and Kim [82] pioneered with the first neural, end-to-end, natural language-to-code search engine. Their model embeds the code of methods using three submodels that apply recurrent neural networks to the name of the method, the API sequences in the method, and all tokens in the method body, respectively. Likewise, the model embeds the words in the query using another recurrent neural network. All embedding models are trained jointly to reduce the distance of matching code-query pairs while keeping unrelated pairs apart. In a similar way, Sun et al. [246] embed a code example and a natural language description into a joint vector space. They improve upon earlier work by translating the code into a natural language-like representation based on transformation rules. Chen and Zhou [34] use two jointly trained auto encoders to map code and text into a vector space, respectively. Cambronero et al. [28] compare different ways to implement neural code search, including unsupervised [222] and supervised approaches and different neural models [82, 102]. Because a single model may not capture all aspects of a code example, Du et al. [48] propose an ensemble model that combines three neural code encoders, which focus on the structure of code, its variables, and its API usages, respectively.

To foster further comparisons, the CodeSearchNet challenge [101] offers a dataset of 2 million pairs of code and natural language queries, along with several neural baseline models and ElasticSearch.[12] Improvements on learning vector representations of code further improve the effectiveness of learning-based code search. For example, learn from multiple code representations [80], apply attention-based neural networks [276], or learn from a graph representation of code and queries via a graph neural network [145].

Learning-based code-to-code search. To find code based on an incomplete code example, several learning-based approaches have been proposed. One approach expands an incomplete code snippet using an LSTM-based language model and then searches for similar code snippets via a scalable clone detection technique [286]. An improved version of the approach [287] uses a library-sensitive language model for expanding the given code snippet. Another approach for retrieving code given an incomplete code snippet learns a model that predicts the probability that a complete code example fits the given snippet [175]. The model is based on various kinds of contextual information, e.g., the types, API calls, and code comments found around the given code snippet.

Search based on pre-trained models. Recent approaches use large pre-trained language models [59, 83], such as BERT [118], for code search. For example, Salza et al. [227] pre-train a BERT model on multiple programming languages and then they fine-tune the model using two encoders: one for natural language queries and another for code snippets. Chai et al. [29] show the value of transfer learning for code search by pre-training CodeBERT [59] on Java and Python, applying a meta-learning approach called MAML (Model-Agnostic Meta-Learning) [60] to adapt the neural model to the target language, and finally fine-tuning the model with a dataset from the target language. Instead of an end-to-end neural search that maps entire code examples and queries into a joint vector space, one can also use pre-trained embeddings of individual words and tokens. For example, Ling et al. [145] use GloVe [198] and Zhou, Zhong, and Shen [286] use pre-trained FastText embeddings.[13] Sachdev et al. [222] propose an approach that maps individual code tokens into vectors, then computes a

[12] `https://www.elastic.co/elasticsearch/`
[13] `https://fasttext.cc`

TFIDF-weighted average of them, and finally uses the resulting vector for a nearest neighbor-based search in the vector space.

Database-based Indexing and Retrieval

Given the success of databases for storing and retrieving information, several code search approaches build upon general-purpose databases. David et al. [40] describe a code search engine for binaries that stores short execution traces ("tracelets") in the NoSQL database MongoDB. Given a function as a query, the approach then retrieves other functions by querying the database for matching tracelets. Another database-based approach is by Hajiyev, Verbaere, and de Moor [86], who build upon a relational database. Their approach stores facts extracted from a program, such as return relationships, method calls, and read and write fields, and then formulates search queries as database queries. In contrast to the similarity-based retrieval techniques discussed above, databases retrieve code examples that precisely match a query.

Graph-based Indexing and Retrieval

Given a graph representation of queries and code, another common approach is to retrieve code via graph-based matching. Li et al. [143] abstract both code snippets and a natural language query into graphs that represent different API method calls and their relationships. Then, they address the retrieval problem as a search for similar graphs. The Yogo search engine [205] represents a given query code example and all code examples to search through as dataflow graphs. To match queries with code examples, the approach then applies a set of rewrite rules to check if the rewritten graphs match.

Instead of matching graphs, another direction is to use a graph representation of code to compute a similarity score. Mcmillan et al. [165]'s Portfolio technique first computes the pairwise similarity of a query and a set of functions, and then propagates the similarity score using the spreading activation algorithm through a pre-computed call graph. In an orthogonal step, the approach also computes the importance of every function by applying the page rank algorithm to the call graph. Finally, the two scores are combined to retrieve relevant functions.

SCRUPLE [195, 196] uses a finite automata-based comparison of a code query and code examples. After turning both into a finite automata, a code pattern automaton interpreter compares two pieces of code and reports a match if the automaton reaches the final state.

Solver-based Matching

Code search engines that represent the behavior of code in the form of constraints often use Satisfiability Modulo Theories (SMT) solvers to match queries against code examples [109, 244]. The indexing phase in this case consists of a static analysis that extracts constraints describing input-output relationships. Then, the retrieval phase checks with an SMT solver whether the constraints of a code example satisfy the input-output examples that a user provides as the query. The idea was first proposed by Stolee, Elbaum, and Dwyer [244] and later refined and generalized by Jiang et al. [109].

Summary: The most used approaches for indexing and retrieval are feature vector-based retrieval and, more recently, deep learning-based models. The first approach needs less data and represents query and source code both as interpretable feature vectors. The second approach needs more data for training a model, e.g., to embed both queries and code source into a joint vector space.

2.4.4 Granularity of Retrieved Source Code

Different code search engines retrieve code at different levels of granularity. We categorize the existing approaches into four kinds of granularity. First, many search engines retrieve *code snippets*, which may range from a single line of code to multiple consecutive lines that implements a specific task. Second, other search engines focus on the *method*-level, i.e., these approaches retrieve entire methods. Third, users can also search at the *class*-level, where code search engines return entire classes. Finally, there also are search engines that operate at the *application or library*-level, which we do not cover in full detail here. Table 2.3 summarizes the approaches and the granularity they use. The same

Table 2.3: Granularity of source code extracted by code search approaches.

Granularity	Approaches
Snippet of code	[12, 15, 29, 30, 31, 34, 40, 44, 62, 82, 99, 124, 143, 146, 151, 154, 172, 175, 184, 189, 195, 196, 205, 222, 227, 236, 242, 246, 256, 266, 282, 286]
Method	[101, 109, 140, 146, 152, 156, 165, 177, 182, 189, 200, 201, 215, 238, 244, 268]
Class	[146, 189, 215]
Application or library	[2, 11, 146, 189]

approach may appear in multiple rows [146, 189] if it supports multiple kinds of granularity.

The design decision of the granularity level to target is very important for a code search engine, because it affects what a user can search for. For example, snippets of code-level can be useful to search for code that provides an example of how to use an API [12]. The disadvantage of retrieving code snippets is that they may be incomplete and thus hard to directly reuse. Searching at the method-level can be useful for finding full methods that already solve a specific task [201], which a user may directly reuse. Finally, class-level and application or library-level approaches are useful to find entire components to reuse. Due to the more coarse-grained granularity, the number of suitable results may be limited though.

2.5 Ranking and Pruning of Search Results

After retrieving code examples that likely match a query, many code search engines rank and prune the results before showing them to the user. This step is critical to enable users to quickly see the most relevant matches. In the following, we discuss and compare different ranking (Section 2.5.1) and pruning (Section 2.5.2) approaches.

2.5.1 Ranking of Search Results

Standard Distance Measures

The by far most common ranking approach is to rely on a distance measure implicitly provided by the retrieval component of a code search engine (Section 2.4). In this approach, the query and each code example are first represented as feature vectors, then a standard distance measure gives the distance between a query vector and a code vector, and finally code examples with a smaller distance to the query are ranked higher. For example, this ranking approach can be implemented using cosine similarity [28, 31, 34, 145, 222, 227, 246, 249], Hamming distance [12], and Euclidean distance [15, 137, 138].

Custom Ranking Techniques

In addition or as an alternative to standard distance measures, several search engines rely on custom ranking techniques. David and Yahav [40] propose a variation of string edit distance to compute the similarity between two sequences of assembly instructions. The basic idea is to treat each instruction as a letter and to use a table that provides a heuristic distance between assembly instructions. Another approach [143] ranks code examples using two scores that are based on the number of tokens that match the given natural language description and the length of a code snippet, respectively. Sachdev et al. [222] augment the rank obtained via cosine similarity with custom rules, such as the number of query tokens present in the candidate, to re-rank the list of results. Another example is from Lu et al. [152]. They compute a representative set of words for each method and then rank results via a normalized intersection of these words.

COSAL [163] combines multiple custom ranking techniques, which compare two pieces of code based on their token similarity, structural similarity, and behavioral similarity, respectively.

Some ranking approaches look beyond the given query by also considering the code a developer is editing while making a query. For example, when building a query vector, Takuya and Masuhara [249] give more weight to occurrences of tokens near the cursor position, to find programs that contain similar fragments to the code around the cursor position. In a similar vein, Wightman et al. [266] uses features of the programmer's source code to rank and filter prospective snippet results, including variable types and names, the cursor position within the abstract syntax tree, and code dependencies. A higher rank here means that a code example uses more of the existing variables etc., and hence will require fewer modifications.

Some more recent ranking approaches are based on machine learning models. For example, the Lancer approach [287] fine-tunes a pre-trained BERT model[14] to predict whether a code example matches the given, incomplete method, and then ranks code examples based on the predicted score. Ye et al. [282] compute the similarity score using two parameters retrieved with a code summarization model and a code generation model, based on a dual learning technique.

Summary: To rank search results, engines often use standard algorithms, such as cosine similarity and Euclidean distance, or they implement custom variations of these techniques.

2.5.2 Pruning of Search Results

Orthogonal to ranking, several search engines also prune search results that are unlikely to be of interest to the user. The most straightforward pruning technique is to discard results based on similarity threshold. For example, some approaches discard all candidates with a similarity lower than some threshold [30, 109], while others show only the top N results in the output [124, 151, 209]. Another way of pruning search results is to merge similar code examples, assuming that

[14]https://github.com/huggingface/pytorch-pretrained-BERT

a user likely wants to see only one of them. For example, Mishne, Shoham, and Yahav [172] merge similar method call paths relevant to the query to remove redundancy in the final results. Aroma [154] uses a greedy algorithm based on parse tree comparison to find and remove redundant code snippets, followed by re-ranking the pruned search results.

Summary: Filtering by a threshold and merging similar results are the most popular techniques for pruning code search results.

2.6 Empirical Studies of Code Search

The wide adoption of code search in practice raises various interesting questions about the way developers search for code. This section discusses empirical studies related to how, when, and why developers search for code and what tools they use for this purpose. We include all such empirical studies that we are aware of and that fit the selection criteria given in Section 2.1. We start by describing

Table 2.4: Overview of empirical studies on code search.

	Singer et al. [235]	Sim, Clarke, and Holt [232]	Ko et al. [126]	Sim et al. [233]	Panchenko, Plattner, and Zeier [192]	Bajracharya and Lopes [13]	Sadowski, Stolee, and Elbaum [223]	Rahman et al. [210]
Topic of study:								
Usage of development tools	✓		✓					
Usage of search tools		✓		✓	✓	✓	✓	✓
Activities of developers	✓	✓	✓				✓	
Methodology:								
Questionnaire	✓	✓	✓				✓	
Log analysis	✓			✓	✓	✓	✓	✓
Observing developers	✓		✓					
Searched code:								
Single project	✓	✓	✓					
Multiple projects within organization	✓				✓		✓	
Many open-source projects				✓		✓		✓

the experimental setups used in these studies (Section 2.6.1) and then present some of their main results (Section 2.6.2). Table 2.4 gives an overview of the discussed studies, including the topics they address, the methodologies they use, and the amount of code searched through by the studied developers.

2.6.1 Setups of Empirical Studies

While practically all empirical studies address the broad questions of how, when, and why developers search for code, they use different setups and methodologies to address this question. Early studies [126, 235] are mostly about what activities developers spend their time on and what tools they use, including tools used for code search. In contrast, more recent studies [13, 192, 210, 223, 232, 233] focus specifically on code search tools and what activities they are used for.

We see three kinds of methodologies, and sometimes combinations of them: questionnaires answered by developers [126, 232, 235], analyses of logs of search engines [13, 192, 210, 223, 233, 235], and observing developers, e.g., by shadowing them [235] or by recording their screens [126]. The first two kinds of studies are typically based on data gathered from tens [126, 232, 233] to hundreds [223] of developers. In contrast, log analysis often covers much larger datasets, ranging between tens of thousands [192] and ten million [13] logged activities.

The studies also vary by the amount of code that is searched through by the studied developers. Reflecting the general trends in code search engines, early studies are about searching through a single project [126, 232, 235], whereas later studies are about searching through multiple projects, either within a larger organization [192, 223] or the open-source ecosystem [13, 210, 233].

2.6.2 Results of Studies and their Implications

A recurring finding in studies is that code search is among the most common activities developers spend their time on. Early studies report that *grep*, *find*, and its variants are used on a regular basis [232, 235]. For example, measurements of tool invocations by Singer et al. [235] shows that *grep* and its variants are the second-most frequently used developer tools, right after the compiler. The observational study by Ko et al. [126] also reports code search to be a common activity. However, their definition of "searching for code" only partially matches ours, because we assume that there is an explicitly formulated query, whereas they also mean reading code to find a specific code location. A study at Google based on a specialized code search engine shows that the average developer is involved in five search sessions per day, with a total of twelve daily queries [223]. Overall, these findings highlight the importance of code search in software development, motivating researchers and practitioners to work on code search techniques.

Several studies investigate the goals that developers have when searching for code. The three most commonly reported goals are finding example code to reuse, e.g., when trying to understanding how to use an API (between 15% [232] and 34% [223] of all searches), program understanding (between 14% [232] and 29% [223] of all searches), and understanding and fixing a bug (between 10% [223] and 20% [232] of all searches). Beyond these three goals, a long tail of other goals is reported, such as understanding the impact of a planned code change, finding locations relevant for a code clean-up, understanding the coding style used within an organization, and identifying the person responsible for a particular piece of code. A perhaps surprising finding is that developers also often use code search as a quick way to navigate through code they are already familiar with [223].

Being a central element of every search, queries and their properties have received some attention in studies. A study of the Koders code search engine finds most queries to be short, with 79% of users providing only a single search term [13]. In contrast, other studies report longer queries, e.g., an average of 4.2 terms per query in a study across five search engines [233], and of 4.7 terms for code-related queries given to Google's general-purpose search engine [210]. Beyond the size of queries, several studies investigate what terms are used in queries. Bajracharya and Lopes [13] find that both code queries and natural language queries are common. Comparing code-related queries with general-purpose web search queries, Rahman et al. [210] find that code queries use a smaller vocabulary. Another interesting finding related to queries is that they are frequently reformulated within a search session [13, 223, 233], even more often than general web search queries [210].

Finally, some studies analyze and compare how effective code search engines are at providing useful search results. One study reports that between 25% and 60% of all queries are effective, depending on the kind of query, where "effective" means that the search results cause the user to download a relevant piece of code [13]. Another study compares specialized code search engines with general-purpose web search engines. It finds that the former are more effective when searching for entire subsystems, e.g., a library to use, whereas the latter are more effective for finding individual blocks of code [233]. The same study also reports that it is easier to find reference examples than components that are reusable as-is.

Summary: Empirical studies of developers show that they commonly perform code search to reach various goals, including code understanding, finding code to reuse, and quickly navigating to code the developer already knows.

2.7 Open Challenges and Research Directions

2.7.1 Support for Additional Usage Scenarios

Each code search engine focuses on one or more usage scenarios, such as finding examples of how to use a specific API or finding again some code that a developer has previously worked on. In addition to the currently supported usage scenarios, we envision future work to support other search-related developer tasks. For example, developers may want to search not only through a static snapshot of code, but also search for specific kinds of changes in the version histories of projects. Searching for changes could help developers, e.g., to understand how a particular API usage typically evolves, to find examples of code changes similar to a change a developer is currently working on, or to find code changes that have introduced bugs. Lawall, Lambert, and Muller [129] and Di Grazia, Bredl, and Pradel [41] propose promising first steps into this direction. Another example of a currently unsupported usage scenario is cross-language search. In this scenario, a user formulates a code query in one programming language to find related code written in another programming language. Such cross-language search could help developers transfer their knowledge across languages, e.g., when a developer knows how to implement a particular functionality in one but not in another language.

2.7.2 Cross-Fertilization with Code Completion and Clone Detection

Code search relates to other problems that have received significant attention by researchers and that offer opportunities for cross-fertilization. One such problem is code completion, i.e., the problem of suggesting suitable code snippets while the developer is writing code in an integrated development environment (IDE). Recent large-scale language models used for code completion offer a functionality similar to code search. For example, a typical usage scenario of GitHub's Copilot tool[15] and the underlying Codex model [33] takes a short natural language description of a desired functionality and maps it to a code snippet offering that functionality. This usage scenario is closely related to code search engines that receive free-form queries (Section 2.2.1). It remains an open challenge to apply successful techniques from code search in code completion, and vice versa. Another problem that is strongly related to code search is clone detection [221]. Similar to code search engines that accept programming language queries (Section 2.2.2), clone detectors try to find code that is similar to a given code example. A key difference is that clone detection tries to find multiple code examples that implement the same functionality (possibly with syntactic and semantic differences that do not affect the overall behavior), whereas code search tries to retrieve code that offers more functionality than the given query. Despite these different goals, there is potential for cross-fertilization of the two related fields, e.g., by adapting effective representations of code (Section 2.4) or mechanisms for pruning search results (Section 2.5.2).

2.7.3 Learning-Based Code Search

Given the tremendous progress in machine learning, adopting the newest models to code search is likely to offer new opportunities to code search. In particular,

[15] https://github.com/features/copilot

we identify three open challenges. First, future work could benefit from the increasingly effective code representation models proposed in the neural software analysis field [202] to compute a vector representation of code and of queries, which can then be used to identify code examples similar to a given query. Some instances of this idea have already been presented [82], but as code representation models keep increasing, adopting new models is likely to also improve code search. Second, future work could design models that not only retrieve code, but also generalizing examples seen during training into new code that fits a query. Open challenges here include to formulate code search as a zero-shot learning problem [24] and to adapt models that combine question answering with retrieval [136].

2.7.4 Deployment and Adoption in Practice

Code search is an area of strong interest by academic researchers, tool builders in industry, and practitioners. Despite the already impressive use of code search by developers, we see several open challenges related to its deployment and adoption in practice. On the one hand, there are challenges faced by people who are running and maintaining a code search engine. For example, the problem of how to incrementally re-index a code corpus when the code is evolving has not yet received significant attention by researchers. A naive approach is to continuously re-index the entire corpus, which is likely to unnecessarily repeat significant computational effort.

On the other hand, there are challenges faced by users of code search engines. While various techniques have been proposed for the core components of code search, its user interface is receiving less attention, with some noteworthy exceptions, such as some of the query expansion techniques discussed in Section 2.3. An interesting line of future work could be to automatically formulate clarification questions, e.g., in natural language, which could allow a user to prune the search space with a single click. Another promising direction is to support users in defining code queries (Section 2.2) by adding automatic code completion features known from IDEs into the interface of a search engine. Finally, future work could design "query-less" search engines that suggest suitable code snippets while a developer is writing code, without the need to explicitly formulate a query. First steps toward that goal have been taken, e.g., by Brandt et al. [22] and Takuya and Masuhara [249].

2.7.5 Common Datasets and Benchmarks

An effective way to foster further progress in the research field is to offer reusable datasets for evaluating and comparing code search engines. Ideally, such a dataset should be realistic, large-scale, and cover multiple programming languages. Several benchmark datasets have been proposed and are use by parts of the existing work. One kind of benchmarks consists of groups of semantically equivalent implementations, e.g. BigCloneBench [248], Google Code Jam[16], and AtCoder[17]. Such benchmarks are particularly useful to evaluate code-to-code search engines (Section 2.2.2), as one implementation in a group can be used as a query, while the other implementations are expected to show up among the results. Benchmarks that come with executable test cases, e.g., those derived from coding competitions, are also useful to evaluate approaches based on dynamic analysis (Section 2.4.1).

Another kind of benchmarks offers pairs of natural language queries and code. For example, the CodeSearchNet challenge offers such a dataset, which has been automatically gathered and covers Go, Java, JavaScript, PHP, Python, and Ruby code [101]. In a similar vein, CodeXGLUE offers query-code pairs for Python and Java [153]. The Search4Code dataset provides code-related queries extracted from Bing search queries via a weakly supervised discriminative model [212]. Instead of relying on automated extraction of datasets, CoSQA is a benchmark of pairs of natural language queries and code examples that have been manually annotated [100].

We anticipate future work to build even more than today upon these datasets, either as a benchmark to evaluate a novel code search engine, or as a training dataset to learn from. There also are opportunities for creating datasets and benchmarks that go beyond those available today. For example, the community would benefit from a dataset that not only includes queries and search results, but also information on how developers act on search results, e.g., by selecting

[16] `https://codingcompetitions.withgoogle.com/codejam`
[17] `https://atcoder.jp/`

lower-ranked results or by copying and adapting code examples. An interesting challenge for benchmarks used to evaluate learning-based code search is how to ensure that a model does not see the benchmark during learning. As large-scale, pre-trained models [33, 59, 83], which often are trained on a large fraction of all publicly available source code, are becoming increasingly popular, the chances that a publicly available benchmark is coincidentally used during training increases.

2.8 Concluding Remarks

This chapter provides a comprehensive overview of 30 years of research on code search. Given the huge amounts of existing code, searching for specific code examples is a common activity during software development. To support developers during this activity, various techniques for finding relevant code have been proposed, with an increase of interest during recent years. We discuss what kinds of queries code search engines support, and give an overview of the main components used to retrieve suitable code examples. In particular, the chapter discusses techniques to pre-process and expand queries, approaches toward indexing and retrieving code, and ways of pruning and ranking search results. Our chapter enables readers to obtain an overview of the field, or to fill in gaps of their knowledge of the state-of-the-art. Based on our survey of past work, we conclude that code search has evolved into a mature research field, with solid results that have already made an impact on real-world software development. Despite all advances, many open challenges remain to be addressed in the future, and we hope this chapter will provide a useful starting point for addressing them.

Chapter 3

DiffSearch: A Scalable and Precise Search Engine for Code Changes

The source code of successful projects is evolving all the time, resulting in hundreds of thousands of code changes stored in source code repositories. This wealth of data can be useful, e.g., to find changes similar to a planned code change or examples of recurring code improvements. This chapter presents DiffSearch, a search engine that, given a query that describes a code change, returns a set of changes that match the query. The approach is enabled by three key contributions. First, we present a query language that extends the underlying programming language with wildcards and placeholders, providing an intuitive way of formulating queries that is easy to adapt to different programming languages. Second, to ensure scalability, the approach indexes code changes in a one-time preprocessing step, mapping them into a feature space, and then performs an efficient search in the feature space for each query. Third, to guarantee precision, i.e., that any returned code change indeed matches the given query, we present a tree-based matching algorithm that checks whether a query can be expanded to a concrete code change. We present implementations for Java, JavaScript, and Python, and show that the approach responds within seconds to queries across one million code changes, has a recall of 80.7% for Java, 89.6% for Python, and 90.4% for JavaScript, enables users to find relevant code changes more effectively than a regular expression-based search and GitHub's search feature, and is helpful for gathering a large-scale dataset of real-world bug fixes.

3.1 Introduction

Hundreds of thousands of code changes are stored in the version histories of code repositories. To benefit from this immense source of knowledge, practitioners and researchers often want to search for specific kinds of code changes. For example, developers may want to search through their own repositories to find again a code change performed in the past, or search for commits that introduce a specific kind of problem. Developers may also want to search through changes in repositories by others, e.g., to understand how code gets migrated from one API to another, or to retrieve examples of common refactorings for educational purposes. A question on Stack Overflow on how to systematically search through code changes[18] has received over half a million views, showing that practitioners are interested in finding changes from the past.

Besides practitioners, researchers also commonly search for specific kinds of code changes. For example, a researcher evaluating a bug finding tool [85] or a program repair tool [134, 174, 250] may be interested in examples of specific kinds of bug fixes. Likewise, researchers working on machine learning models that predict when and where to apply specific code changes require examples of such changes as training data [10]. Finally, researchers systematically study when and how developers perform specific kinds of changes to increase our understanding of development practices [52, 176, 179, 211].

Unfortunately, there currently is no efficient and effective technique for systematically searching large version histories for specific kinds of changes. The solutions proposed in the above Stack Overflow post are all based on matching regular expressions against raw diffs. However, searching for anything beyond the most simple change patterns with a regular expression is cumbersome and

[18]https://stackoverflow.com/questions/2928584/how-to-grep-search-committed-code-in-the-git-history

likely to result in irrelevant code changes. Another existing technique is GitHub Search,[19] which allows for searching through commits using free-form queries that are matched, e.g., against commit messages. However, both regular expressions and GitHub Search have significant drawbacks when searching for specific code changes, as we show in a user study. Finally, previous research proposes techniques that linearly scan version histories for specific patterns [61, 116, 131, 191]. However, due to their linear design, these techniques do not scale well to searching through hundreds of thousands of changes in a short time.

This chapter presents DiffSearch, a scalable and precise search engine for code changes. DiffSearch is enabled by three key contributions. First, we design a query language that is intuitive to use and easy to adapt to different programming languages. The query language extends the target programming language with wildcards and placeholders that abstract specific syntactic categories, e.g., expressions. Second, to ensure scalability, the approach is split into an indexing part, which maps code changes into a feature space, and a retrieval part, which matches a given query in the feature space. We design specific features for code changes, extracting useful information to match different changes on source code. Finally, to ensure precision, i.e., that a found code change indeed fits the given query, a crucial part of the approach is to match candidate code changes against the given query. We present an efficient algorithm that checks if a query can be expanded into a code change.

Our approach supports the different usage scenarios we envision DiffSearch to be useful for. First, the approach supports users interested in finding *one* specific code change, e.g., when searching through the history of their own project to find some change done by a colleague. In this scenario, similar to a classical web search engine, the user will consider only the first few search results and stop inspecting them as soon as the expected code change is found. Second, DiffSearch supports users interested in finding *multiple* code changes, e.g., when searching through a set of popular open-source projects to find examples of typical ways to refactor a specific API usage. In this scenario, the user will inspect the ranked list of search results until having seen a sufficient number of examples. Third, the approach supports users interested in finding *many*

[19] `https://github.com/search`

code changes, e.g., to build a large-scale dataset to train a neural model. In this scenario, the user can formulate and fine-tune the query through the interactive user interface of DiffSearch, and then download all matching results at once into a file. Finally, DiffSearch can also be configured to retrieve *all* code changes that match a query, e.g., to quantify how often specific changes occur in practice. In this scenario, the user turns off the indexing and retrieval part of the approach, and instead runs the precise matching of a query against all code changes.[20]

DiffSearch is designed in a mostly language-agnostic way, making it possible to apply the approach to different languages. In particular, we restrict ourselves to a very lightweight static analysis of code changes. The query language and parts of the search algorithm build upon the context-free grammar of the target programming language. As a proof-of-concept, DiffSearch currently supports three widely used languages: Java, JavaScript, and Python.

Our approach relates to work on searching for code, which retrieves code snippets that match keywords [11, 82], test cases [215], or partial code snippets [124, 154]. While code search engines often have a design similar to ours, i.e., based on indexing and retrieval, they consider only a single snapshot of code, but not code changes. Other related work synthesizes an edit program from one or more code changes [56, 58, 61, 64, 219] and infers recurring code change patterns [179, 190]. Starting from concrete changes, these approaches yield abstractions of them. Our work addresses the inverse problem: given a query that describes a set of code changes, find concrete examples that match the query. Finally, our work relates to clone detection [107, 114, 144, 220, 224], as DiffSearch searches for code changes that resemble a query. Our work differs from clone detection by considering code changes (and not individual snippets of code), by focusing on guaranteed matches instead of similar code, and by responding to queries quickly enough for interactive use.

We evaluate the effectiveness and scalability of DiffSearch with one million code changes in each of Java, Python, and JavaScript. We find that the approach responds to queries within a few seconds, scaling well to large sets of code changes. The search has a mean recall of 80.7% for Java, 89.6% for Python, and 90.4% for JavaScript, which can be increased even further in exchange for a

[20]As shown in the evaluation, guaranteeing to find *all* matching code changes comes at the cost of efficiency, as it requires a linear search through all code changes in the corpus.

slight increase in response time. A user study shows that DiffSearch enables users to effectively retrieve code changes, clearly outperforming a regular expression-based search through raw diffs and GitHub Search. As a case study to show the usefulness of DiffSearch for researchers, we apply the approach to gather a dataset of 74,903 bug fixes.

In summary, this chapter contributes the following:

- A *query language* that extends the target programming language with placeholders and wildcards, making it easy to adapt the approach to different languages.
- A technique for searching for code changes that ensures *scalability* through approximate, indexing-based retrieval, and that ensures *precision* via exact matching.
- Empirical evidence that the approach effectively finds thousands of relevant code changes, scales well to more than a million changes from different projects, and successfully helps users answer a diverse set of queries.

The implementation and a web interface of DiffSearch are publicly available:

http://diffsearch.software-lab.org

3.2 Example and Overview

3.2.1 Motivating Example

To illustrate the problem and how DiffSearch addresses it, consider the following example query. The query searches for code changes that swap the arguments passed to a call that is immediately used in a conditional. Such a query could be used to find fixes of swapped argument bugs [217].

```
if(ID<1>(EXPR<1>, EXPR<2>)){    →  if(ID<1>(EXPR<2>, EXPR<1>)){
  <...>                              <...>
```

Our query language is an extension of the target programming language, Java in the example, and adds placeholders for some syntactic categories. For example, the `ID<1>` placeholder matches any identifier, and the `EXPR<1>` placeholder matches any expression. Instead of such placeholders, queries can also include concrete identifiers and literals, e.g., to search for specific API changes.

As the set of code changes to search through, suppose we have the following three examples, of which only the second matches the query:

Code change 1:

```
if(check(a - 1, b)){    →   if(check(a - 1, c)){
```

Code change 2:

```
if(isValidPoint(x, y)){   → if(isValidPoint(y, x)){
```

Code change 3:

```
while(var > k - 1){     → while(var > k){
  sum += count(var);         sum += 2 * count(var);
```

3.2.2 Problem Statement

An important design decision is the granularity of code changes to consider. The options range from changes of individual lines, which would limit the approach to very simple code changes, to entire commits, which may span multiple files, several dozens of lines [3], often containing multiple entangled logical changes [16, 94, 116, 193]. We opt for a middle ground between these two extremes and consider code changes at the level of "hunks", i.e., consecutive lines that are added, modified, or removed together.

Definition 3.1 (Code change)
A code change $c \rightarrow c'$ consists of two pieces of code, each of which is a sequence $[l_1, .., l_m]$ of consecutive lines of code extracted from a file in the target language.

Definition 3.2 (Query)
A query $q \rightarrow q'$ consists of two patterns, which each are a sequence $[l_1, .., l_m]$ of lines of code in an extension of the target programming language. The language extension adds wildcards, a special "empty" symbol, and placeholders for specific syntactic categories, e.g., to match an arbitrary expression or identifier.

Given these two ingredients, the problem we address is:

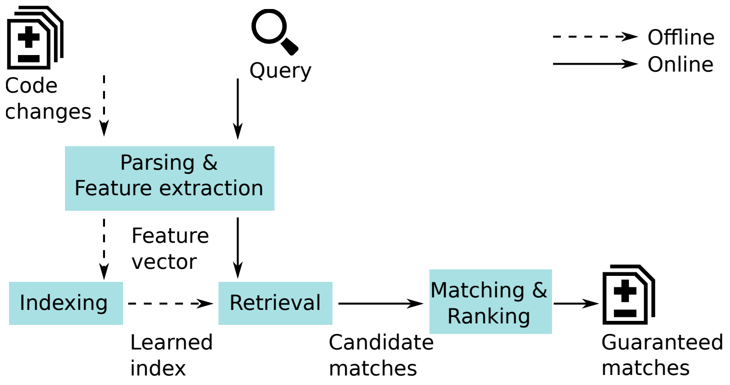


Figure 3.1: Overview of the approach.

Definition 3.3 (Search for code changes)
Given a set C of code changes and a query $q \rightarrow q'$, find a set $M \subseteq C$ of code changes such that each $(c \rightarrow c') \in M$ matches $q \rightarrow q'$. We say that a code change $c \rightarrow c'$ matches a query $q \rightarrow q'$ if there exists an expansion of the placeholders and wildcards in $q \rightarrow q'$ that leads to $c \rightarrow c'$.

By ensuring that, for any retrieved code change, the query can be expanded to the code change, DiffSearch guarantees that every result of a search precisely matches the query.

3.2.3 Main Idea of the Approach

DiffSearch consists of four components that are used in an offline and an online phase as illustrated in Section 3.1. In the offline phase, the approach analyzes and indexes a large set of code changes. The *Parsing & Feature extraction* component of the approach parses and abstracts concrete code changes and queries into a set of features, mapping both into a common feature space. For our example query in Section 3.2.1, the features encode, e.g., that a call expression appearing within the condition of an if statement is changed and that the changed call has two arguments. To enable quickly searching through hundreds of thousands of code changes, the *Indexing* component of DiffSearch indexes the given feature vectors [111] once before accepting queries.

In the online phase, the input is a query that describes the kind of code changes to find. Based on the pre-computed index and the feature vector of a given query, the *Retrieval* component retrieves those code changes that are most similar to the query. For our motivating example, this yields Code change 1 and Code change 2 because both change the arguments passed to a call. The similarity-based retrieval does not guarantee precision, i.e., that each candidate code change indeed matches the query. The *Matching & Ranking* component of DiffSearch removes any candidates that do not match the query by checking whether the placeholders and wildcards in the query can be expanded into concrete code in a way that yields the candidate code change. For our example, matching will eliminate Code change 1, as it does not swap arguments, and eventually returns Code change 2 as a search result to the user.

3.3 Approach

This section presents the approach in detail. Before going through the four components introduced in Section 3.2.3, we define the query language to specify what kind of code changes to search for.

3.3.1 Query Language

To search for specific kinds of code changes, DiffSearch accepts queries that describe the code before and after the change. Our goal is to provide a query language that developers can learn with minimal effort and that supports all constructs of the target programming language. We initially considered three possible kinds of code search queries, as classified by Di Grazia et al. [42]. First, natural language queries, which are easy to type but inherently imprecise. Second, programming language queries, which require knowing the programming language and are precise. Third, custom languages that are often the most precise, but they may impose some effort to learn the new language [42].

Comparing the different options and considering the envisioned users of our approach, we design the query language of DiffSearch as an extension of the target programming language. That is, the query language includes all rules of

Query	::=	*Snippet* → *Snippet*
Snippet	::=	*Stmt** | *Expression* | _
Stmt	::=	⟨...⟩ | (Target language rules)
Expression	::=	EXPR | EXPR⟨*Number*⟩ | ⟨...⟩ | (Target language rules)
AssignOperator	::=	OP | OP⟨*Number*⟩ | (Target language rules)
BinaryOperator	::=	binOP | binOP⟨*Number*⟩ | (Target language rules)
UnaryOperator	::=	unOP | unOP⟨*Number*⟩ | (Target language rules)
Identifier	::=	ID | ID⟨*Number*⟩ | (Target language rules)
Literal	::=	LT | LT⟨*Number*⟩ | (Target language rules)

Figure 3.2: Simplified grammar of queries. Non-terminals are in *italics*.

the target programming language and additional features useful for queries. As our approach can support different target languages, this means that there is a different query language for each target language, each extending the target language with search-related keywords. That is, a user who is already familiar with the target programming language needs to learn only a handful of new keywords for using DiffSearch.

Section 3.2 shows the grammar of our query language. A query consists of two sequences of statements, which describe the old and new code, respectively. The syntax for statements is inherited from the target programming language and not shown in the grammar. Instead of a regular code snippet, a query may contain an underscore to indicate the absence of any code, which is useful to describe code changes that insert or remove code. The grammar extends the target language by adding placeholders for specific syntactic entities, namely expressions, operators, identifiers, and literals. For each such entity, a query can either describe with an unnamed placeholder that there should be any such entity, e.g., `EXPR` for any expression, or repeatedly refer to a specific entity with a named placeholder, e.g., using `EXPR<1>` and `EXPR<2>`. Named placeholders will be bound to the same entity across the entire query, e.g., to say that the same expression `EXPR<1>` must appear on both sides. We also introduce the wildcard `<...>` that matches any statement, any expression, or nothing at all.

To illustrate the query language, Table 3.1 gives a few examples of code changes and a corresponding query that matches the code change. The first two

Table 3.1: Examples of Java changes and matching queries.

Code change	DiffSearch query		
`- evt.trig();`	`ID.ID();`	→	`_`

`- if (x > 0)`
`-   y = 1;`
`+ if (x < 0)`
`+   y = 0;`
	`if (EXPR)`
	`  ID OP LT;`
		→	`if (EXPR)`
			`  ID OP LT;`

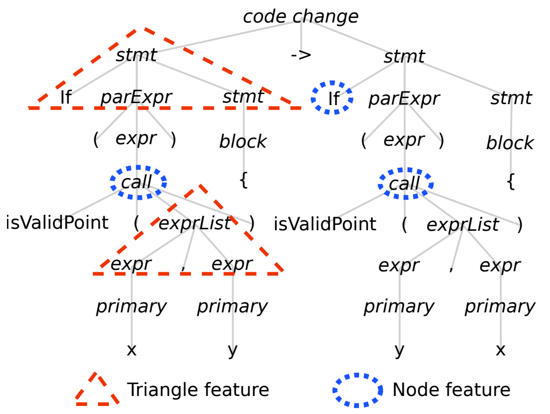
`- run(k);`
`- now(k);`
`+ runNow(k);`
	`run(EXPR<0>);`
	`now(EXPR<0>);`
		→	`runNow(EXPR<0>);`

examples use unnamed placeholders, e.g., to match arbitrary identifiers. The third example uses a named placeholder: The `EXPR<0>` in both the old and new part of the query means that this expression, here `k`, remains the same despite the code change, which replaces two calls with one.

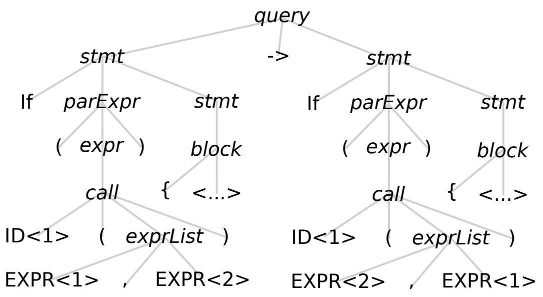
3.3.2 Tree-based Representation of Code Changes and Queries

One goal of DiffSearch is to be mostly language-agnostic, making it possible to apply the approach to different programming languages. Our current version supports Java, JavaScript, and Python. To this end, the approach represents code changes and queries using a parse tree, i.e., a representation that is straightforward to obtain for any programming language. The benefit of parse trees is that they abstract away some details, such as irrelevant whitespace, yet provide an accurate representation of code changes.

To represent a set of commits in a version history as pairs of trees, DiffSearch first splits each commit into hunks, which results in a set of code changes (Definition 3.1). The approach then parses the old and new code of a hunk using the programming language grammar into a single tree that represents the code change. Likewise, to represent a query, DiffSearch parses the query into a parse tree using our extension of the grammar (Figure 3.2). For example, Figure 3.3 shows the parse trees of a change and a query. The code change (a) corresponds to Code change 2 from Section 3.2, which swaps `x` and `y` of a call to `isValidPoint`. Note that code edits that do not cause any change of the parse



(a) Code change.



(b) Query.

Figure 3.3: Parse tree representations of Code change 2 (a) and the query from Section 3.2 (b). Only some of all considered features are highlighted for illustration.

tree, e.g., because only semantically irrelevant whitespace gets changed, are not considered as code changes and ignored by DiffSearch.

An interesting challenge in parsing code changes and queries is syntactically incomplete code snippets. For example, the code changes in Section 3.2 open a block with { but do not close it with }, because the line with the closing curly brace was not changed. DiffSearch addresses this challenge by relaxing the grammar of the target language so that it accepts individual code lines even when they are syntactically incomplete. For example, we relax the grammar to allow for unmatched parentheses and partial expressions.

As a potential alternative to parse trees, we considered and eventually decided against abstract syntax trees (ASTs). While ASTs are a suitable representation, e.g., for compilers, they abstract away too many syntactic details that may be relevant in DiffSearch. For example, consider the following code change that adds parentheses to make a complex expression easier to read:

```
flag = alive || x && y;  →   flag = alive || (x && y);
```

Because the added parentheses preserve the semantics of the expression, they are abstracted away in a typical AST, i.e., the old and new code have the same AST. As a result, an AST-based representation could neither represent this change nor a query to search for it.

3.3.3 Extracting Features

Based on the tree representation of code changes and queries, the feature extraction component of DiffSearch represents each tree as a set of features. The goal of this step is to enable quickly searching through hundreds of thousands of code changes. By projecting both code changes and queries into the same feature space, we enable the approach to compare them efficiently. An alternative would be to pairwise compare each code change with a given query [61, 131]. However, such a pairwise comparison would require an amount of computation time that is linear w.r.t. the number of code changes, which would negatively affect the efficiency of searching through many code changes.

DiffSearch uses two kinds of features. The first kind of feature is *node features*, which encode the presence of a node in the parse tree. For the example in Section 3.3, the dotted, blue lines show three of the extracted node features. The second kind of feature is *parse tree triangles*, which encode the presence of a specific subtree. Each parse tree triangle is a tree that consists of a node and all its descendants up to some configurable depth. We use a depth of one as a default, i.e., a triangle contains a node and its immediate child nodes. For the example in Section 3.3, the dashed, red lines highlight two of the extracted triangles. The triangle at the top encodes the fact that there is an if statement,

Algorithmus 3.1 Represent features as fixed-size vector.

```
Input: Set $F$ of features, target size $l_{target}$
Output: Feature vector $v$
1: $v \leftarrow$ vector of $l_{target}$ zeros
2: for all $f \in F$ do
3:     $h \leftarrow hash(f)$
4:     $v[h \bmod l_{target}] \leftarrow 1$
5: end for
6: return $v$
```

while the other triangle encodes the fact that the code contains an expression list with exactly two expressions. The two kinds of features complement each other because node features encode information about individual nodes, including identifiers and operators, whereas parse tree triangles represent how nodes are connected.

For each code change or query, the approach extracts a separate set of features for the old and the new code. With this separation, the features encode whether specific code elements are added or removed in a code change. The feature sets for code changes and queries are constructed in the same way, except that DiffSearch removes node features for placeholder nodes, e.g., `ID` or `EXPR`, from the query. The rationale is that we want the features of a query to be a subset of the features of a matching code change, but placeholder nodes never appear in code changes.

Different code changes and queries yield different numbers of features. To efficiently compare a given query against arbitrary code changes, DiffSearch represents all features of a code change or query as a fixed-size feature vector. The feature vector is a binary vector of length $l_n + l'_n + l_{tri} + l'_{tri} = l$, where l_n and l'_n are the number of bits to represent the node features of the old and new code, respectively, and likewise for l_{tri} and l'_{tri} for the parse tree triangle features. We use $l = 1{,}000$ by default, dividing it equally among the four components, which strikes a balance between representing a diverse set of features and efficiency during indexing and retrieval. Section 3.5.5 evaluates different sizes for the feature vector length.

Algorithm 3.1 summarizes how DiffSearch maps a set F of features into a fixed-size vector v. The algorithm computes a hash function over the string representations of individual nodes in a feature, sums up the hash values into a value h, and sets the h-th index of the feature vector to one. To ensure that the index is within the bounds of v, line 4 performs a modulo operation. For each code change or query, the algorithm is invoked four times to map each of the four feature sets into a fixed-size vector: parent-child and triangle features, for both the old and new code.

3.3.4 Indexing and Retrieving Code Changes

To prepare for responding to queries, DiffSearch runs an offline phase that indexes the given set of code changes. The indexing and retrieval components of the approach build on FAISS, which is prior work on efficiently searching for similar vectors across a large set of vectors [111]. In the first step of the offline phase, DiffSearch parses all code changes and stores the parse trees on disk. In the second step, DiffSearch generates the feature vectors of the code changes using the corresponding parse trees. Given the set $V_{changes}$ of feature vectors of all code changes, the approach computes an index into these vectors.

After the offline indexing phase, DiffSearch accepts queries. For a given query, the approach computes a feature vector v_{query} (Section 3.3.3), and then uses the index to efficiently retrieve the most similar feature vectors of code changes. FAISS allows for efficiently answering approximate nearest neighbor queries, without comparing the query against each vector in $V_{changes}$. The nearest neighbors are based on the L2 (Euclidean) distance. To ensure that the presence of matching features is weighted higher than the absence of features, we multiply v_{query} by a constant factor $\frac{l}{2}+1$ before running the nearest neighbor query. To illustrate this decision consider an example with three feature vectors: A query $v_Q = (0,0,1)$, a potential match $v_P = (1,1,1)$ with the third feature in common, and a mismatch $v_M = (0,0,0)$. Naively computing the Euclidean distances yields $d(v_Q, v_P) = \sqrt{2}$ and $d(v_Q, v_M) = \sqrt{1}$, i.e., the mismatch would be closer to the query than the potential match. To avoid this scenario, the query vector should be $v_Q = (0,0,m)$ such that $d(v_Q, v_P) < d(v_q, v_M)$. Solving this inequality gives $m > \frac{l}{2}$, which we achieve by multiplying the original v_Q with $\frac{l}{2}+1$. For the example, after multiplying v_Q with the constant factor $\frac{3}{2}+1$, we have $d(v_Q, v_P) = \sqrt{4.25}$

and $d(v_Q, v_M) = \sqrt{6.25}$, i.e., the potential match is now closer to the query than to the mismatch.

The approach retrieves the k most similar code changes for a given query. Setting the value of k allows users to control the trade-off between efficiency and recall. For example, if a user is interested in finding as many code changes as possible, a larger k should be used. In the extreme case, DiffSearch can also be used without the feature-based retrieval (equivalent to $k = \infty$), which will reduce the approach to linearly searching through all code changes, but guarantees to find each matching code change. We use $k = 5{,}000$ by default, and Section 3.5.5 evaluates other values. The retrieved candidate code changes are ranked based on their L2 distance to the query, computed by FAISS, and we use this ranking to sort the final search results shown to a user.

3.3.5 Matching of Candidate Search Results

Given the k candidate code changes retrieved for a given query as described in Section 3.3.4, DiffSearch could return all of them to the user. However, the feature-based search does not guarantee precision, i.e., that all the retrieved code changes indeed match the query. One reason is that the features capture only local information, but do not encode the entire parse tree in a lossless way. Another reason is that the features do not encode the semantics of named placeholders, i.e., they cannot ensure that placeholders are expanded consistently across the old and new code.

To guarantee that all code changes returned in response to a query precisely match the query, the matching component of DiffSearch takes the candidate search results obtained via the feature-based retrieval and checks for each candidate whether it indeed matches the query. Intuitively, a code change matches a query if the placeholders and wildcards in the query can be expanded in a way that yields code identical to the code change or some subset of the code change. More formally, we define this idea as follows:

Definition 3.4 (Match)
Given a code change $c \rightarrow c'$ and a query $q \rightarrow q'$, let $t_c, t_{c'}, t_q, t_{q'}$ be the corresponding parse trees. The code change matches the query if

- *t_q can be expanded into some subtree of t_c and*

- *$t_{q'}$ can be expanded into some subtree of $t_{c'}$*

so that all of the following conditions hold:

- *Each placeholder is expanded into a subtree of the corresponding syntactic entity.*
- *All occurrences of a named placeholder are consistently mapped to identical subtrees.*
- *Each wildcard is expanded to an arbitrary, possibly empty subtree.*

For example, consider the query and code change in Figure 3.3 again. They match because the code change tree (a) can be expanded into the query tree (b). The expansion maps the named placeholders `ID<1>` to `isValidPoint`, `EXPR<1>` to the subtree that represents `x`, and `EXPR<2>` to the subtree that represents `y`. Moreover the wildcards in the query are both mapped to the empty tree. As an example of a code change that does not match this query, consider Code change 1 from Section 3.2 again. The parse tree of the query cannot be expanded into the parse tree of that code change because there is no way of expanding the query tree while consistently mapping `EXPR<1>` and `EXPR<2>` to the three method arguments `a-1`, `b`, and `c`.

To check whether a candidate code change indeed matches the given query, DiffSearch compares the parse tree of the query with the parse tree of the code change in a top-down, left-to-right manner. The basic idea is to search for a mapping of nodes in the query tree to nodes in the parse tree that consistently maps named placeholders to identical subtrees. On top of this basic idea, the matching algorithm faces two interesting challenges. We illustrate the challenges with the following query, which searches for code changes where two call statements get replaced by an assignment of a literal to an identifier. The following example shows the query on the left and a matching code change on the right:

```
ID();                              foo();     x = 5;
<...>    →    ID = LT;             bar(); →   foo();
ID();                              baz();     y = 7;
```

The first challenge is because queries are allowed to match parts of a change, which is useful to find relevant changes surrounded by other, irrelevant changed

Algorithmus 3.2 Check if a code change matches a query.

```
Input: Code change $c \rightarrow c'$ and query $q \rightarrow q'$
Output: True if they match, False otherwise.
 1: $t_c, t_{c'} \leftarrow parse(c \rightarrow c')$
 2: $t_q, t_{q'} \leftarrow parse(q \rightarrow q')$
 3: $N_{toMatch} \leftarrow (allNodes(q) \cup allNodes(q')) \setminus wildcards$
 4: $W \leftarrow candidateMappings(t_c, t_{c'}, t_q, t_{q'})$
 5: while $W$ is not empty do
 6:     $M \leftarrow$ Take a mapping from $W$
 7:     $n_q \leftarrow nextUnmatchedNode(M, t_q, t_{q'})$
 8:     $n_{pq} \leftarrow$ Parent of $n_q$
 9:     $n_{pc} \leftarrow$ Look up $n_{pq}$ in $M$
10:     for $c$ in all not yet matched children of $n_{pc}$ do
11:         if $canAddToMap(M, c, n_q)$ then
12:             $M' \leftarrow$ Copy of $M$ with $n_q \mapsto c$
13:             if $keys(M') \cap N_{toMatch} = \emptyset$
14:               and $isValid(M, t_c, t_{c'}, t_q, t_{q'})$ then
15:                 return true
16:             end if
17:         else
18:             Add $M'$ to $W$
19:         end if
20:     end for
21: end while
```

code. While useful, this property of queries also implies that the query may match at multiple places within a given code change. In the above example, the `ID = LT;` part of the query may match both `x = 5;` and `y = 7;`. The second challenge is because queries may contain wildcards (`<...>`), which is useful to leave parts of a query unspecified. Wildcards can match none, one, or multiple statements or expressions, and hence, they may cause a single query to match in multiple ways. For the above example, the wildcard could be between the calls of `foo` and `baz`, between the calls of `foo` and `bar`, or between the calls of `bar` and `baz`. Because of these two challenges, matching must consider different ways of mapping a query onto a code change, which results in a search space of possible matches that must be explored.

DiffSearch addresses these challenges in Algorithm 3.2, which checks whether a given query and code change match. The algorithm starts by parsing the code change into trees t_c and $t_{c'}$, which represent the old and new part of the change, and likewise for the query. The core of the algorithm is a worklist-based search through possible mappings between nodes in the parse tree of the query and nodes in the parse tree of the code change. These mappings are represented as a map M from nodes in the query trees to nodes in the code change trees. Each mapping M in the worklist W represents a possible way of matching the query against the code change. To determine whether all nodes in the query have been successfully mapped, the algorithm maintains a set $N_{toMatch}$ of all the nodes in the query that must be matched. The algorithm explores mappings in W until it either finds a mapping that covers all nodes in $N_{toMatch}$, or until it has unsuccessfully explored all mappings in W.

Algorithm 3.2 relies on several helper functions. One of them, *candidateMappings*, computes the starting points for the algorithm by returning all possible mappings of the roots of t_q and $t_{q'}$ to nodes in the code change trees. The *nextUnmatchedNode* function performs a top-down, left-to-right pass through the query trees to find a node that is not yet in the current map M. The *canAddToMap* function checks if adding a mapping $n_q \mapsto c$ is consistent with an already existing map M. Specifically, it checks that n_q is not yet among the keys of M, that c is not yet among the values of M, and that the two nodes are either identical non-placeholder nodes or that n_q is a placeholder that can be consistently mapped to c as specified in Definition 3.4. Finally, the helper function *isValid* checks whether a mapping M that covers all to-be-matched nodes ignores nodes in the change tree only when there is a corresponding wildcard in the query tree. The algorithm postpones this check to *isValid* to reduce the total number of mappings to explore.

Matching a single code change against a query might cause the algorithm to explore many different mappings, and DiffSearch typically invokes Algorithm 3.2 not only once but for tens or hundreds of candidate search results. To ensure that the approach responds to queries quickly enough for interactive usage, we optimize Algorithm 3.2 by pruning code changes that certainly cannot match a given query. To this end, the approach checks if all leaf nodes in the parse tree of a query occur at least once in the parse tree of the code change. For example, consider the following query, which searches for changes in the right-hand side of assignments to a variable `myVar`:[21]

```
myVar = LT;    →   myVar = LT;
```

If a code change does not include any token `myVar`, then the optimization immediately decides that the code change cannot match the query and skips Algorithm 3.2, similar to Coccinelle [130].

3.4 Implementation

We implement the DiffSearch idea in a practical search engine that supports multiple programming languages, currently Java, JavaScript, and Python. To gather raw code changes, the implementation uses ”git log -p”. For each change, a parse tree is created using ANTLR4,[22] using the grammar of the target programming language, modified to support queries and to allow for syntactically incomplete code fragments (Section 3.3.1). The indexing and retrieval components build on the FAISS library [111], which supports efficient vector similarity queries for up to billions of vectors. Once changes are indexed, the search engine is a server that responds to queries via one of two publicly available interfaces: a web interface for interactive usage and a web service for larger-scale usage, e.g., to create a dataset of changes.[23]

[21]Because the `myVar =` part of the code remains the same, the query expresses that the literal captured by the unnamed placeholder `LT` is changing.

[22]`https://www.antlr.org/`

[23]`http://diffsearch.software-lab.org`

3.5 Evaluation

Our evaluation focuses on six research questions:

- RQ1: What is the recall of DiffSearch? (Section 3.5.1)
- RQ2: How efficient and scalable is DiffSearch? (Section 3.5.2)
- RQ3: Does DiffSearch enable users to find relevant code changes more effectively than a regular expression-based search through raw diffs? (Section 3.5.3)
- RQ4: Is DiffSearch useful for finding examples of recurring bug fix patterns? (Section 3.5.4)
- RQ5: How do parameters of the approach influence the results? (Section 3.5.5)
- RQ6: How do queries and search results compare in terms of their size and absolute number? (Section 3.5.6)

For each of RQ1, RQ2, RQ5, and RQ6, we present results for all three currently supported target languages: Java, JavaScript, and Python. For each language, we gather at least one million code changes from repositories that are among the top 100 of their language based on GitHub stars. We compute the average size of the code change pair (old code and new code) in these datasets. The datasets do not contain commit messages, meta-information or code context, but only the removed and added lines, as represented in the diff. As a result, we count the number of '\n' in each pair using the bash command "grep -o '\n' dataset | wc -l" and we find an average number of lines per each pair of 13.4, 8.2, and 7.3 for Java, Python and JavaScript, respectively. For RQ3 and RQ4, we focus on Java as the target language because RQ3 is based on a user study and because RQ4 builds on a Java dataset created by prior work [115]. The experiments are performed on a server with 48 Intel Xeon CPU cores clocked at 2.2GHz, 250GB of RAM, running Ubuntu 18.04.

3.5.1 RQ1: Recall

While the precision of DiffSearch's results is guaranteed by design (Section 3.3.5), the approach may miss code changes due to its feature-based search, which ensures scalability but may fail to include an expected code change into the candidate matches. Additionally, DiffSearch only considers k candidate changes, so it can find at most k results even though queries could have more than k matching code changes.

To establish a ground truth, we randomly sample code changes $c \rightarrow c'$ from all indexed Java, Python, and JavaScript code changes and formulate a corresponding query $q \rightarrow q'$ using the following four strategies. The *as-is* strategy simply copies c into q and c' into q'. The *less-placeholders* strategy replaces some of the identifiers, operators, and literals with corresponding placeholders or wildcards. The *more-placeholders* strategy, similarly, replaces the majority of the identifiers, operators, and literals. Finally, the *generalized* strategy replaces most or all of the identifiers, operators, and literals. For each strategy and each programming language, we randomly sample 20 code changes and construct a query for each one. We then compare each query against all 1,001,797 Java, 1,007,543 JavaScript, and 1,016,619 Python code changes using the matching component of DiffSearch. While significantly slower than the feature-supported search that DiffSearch uses otherwise, this approach allows us to determine the set of all code changes expected to be found for a query, because Algorithm 3.2 precisely computes whether a code change matches a query. By design of DiffSearch (Section 3.3.5) and the way we construct the ground truth, the precision and the mean reciprocal rank (MRR) are 100% and 1.0, respectively, and we hence do not report them in Table 3.2.

Table 3.2 shows the recall of DiffSearch w.r.t. the ground truth, i.e., the percentage of all ground truth code changes that the approach finds. On average across the 80 queries per programming language, DiffSearch has a recall of 80.7% for Java, 89.6% for Python, and 90.4% for JavaScript. More specific queries tend to lead to a higher recall. The reason is that the parse tree of a more generalized query shares fewer features with a matching code change, e.g., because a complex subtree is folded into an `EXPR` node. The slightly higher recall for Python and JavaScript can be explained by two observations. First,

Table 3.2: Recall of DiffSearch across 80 queries per language.

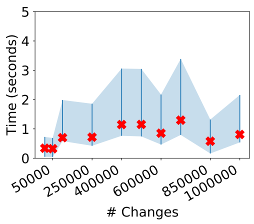
Queries	Java	Python	JavaScript
As-is	90.6%	100.0%	100.0%
Less-placeholders	83.5%	99.9%	99.8%
More-placeholders	74.2%	96.7%	95.8%
Generalized	76.7%	74.9%	66.1%
Total	**80.7%**	**89.6%**	**90.4%**

code changes in Java tend to be slightly larger, causing more nodes on the parse trees, which reduces the chance to find a suitable candidate change, e.g. because the probability of hash collisions is higher if there are more features. Second, across the 80 queries, there are 236,836 ground truth code changes for Java, but only 69,626 and 59,789 for Python and JavaScript, respectively, making finding all ground truth code changes in Java a harder problem. We discuss in Section 3.5.5 that the recall can be increased even further by retrieving more candidate matches, at the expense of a slightly increased response time.

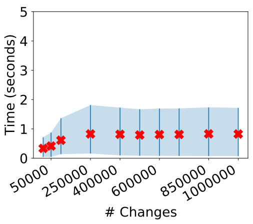
3.5.2 RQ2: Efficiency and Scalability

A major goal of this work is to enable quickly searching through hundreds of thousands of code changes. The following evaluates how the number of code changes to search through influences the efficiency of queries, i.e., how well DiffSearch scales to large amounts of changes. As queries to run, we use the 80 queries described in Section 3.5.1. For each query, we measure how long DiffSearch takes to retrieve code changes from ten increasingly large datasets, ranging from 10,000 to 1,000,000 code changes.

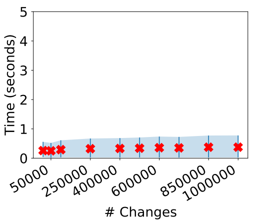
The top row of Figure 3.4 shows the results for the full DiffSearch approach. Answering a query typically takes between 0.5 and 2 seconds. Moreover, the response time remains constant when searching through more code changes. The reasons are (i) that FAISS [111] provides constant-time retrieval in the vector space, and (ii) that the time for matching candidate changes against the query is proportional to the constant number *k* of candidate changes. Comparing the three programming languages, we find that they yield similar performance



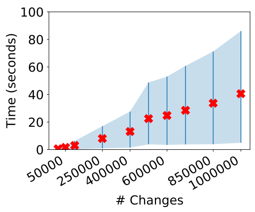
(a) DiffSearch (Java).



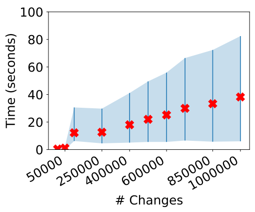
(b) DiffSearch (Python).



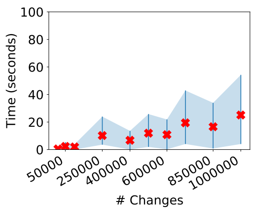
(c) DiffSearch (JavaScript).



(d) DiffSearch without indexing (Java).



(e) DiffSearch without indexing (Python).



(f) DiffSearch without indexing (JavaScript).

Figure 3.4: Response time across differently sized datasets (average and 95% confidence interval). Top: Full DiffSearch. Bottom: DiffSearch without indexing.

results, which is due to the fact that most parts of our implementation are language-agnostic. We conclude that DiffSearch scales well to hundreds of thousands of changes and remains efficient enough for interactive use.

The bottom row of Figure 3.4 shows the same experiment when removing the indexing and retrieval steps of DiffSearch (note: different y-axis). Instead, the approach linearly goes through all code changes and compares them against a given query using the matching component only. Answering a query takes up to 41 seconds on average, showing that the feature-based indexing is essential to ensure DiffSearch's scalability.

Even though scalability is most relevant for the online part of DiffSearch, we also measure how long the offline part takes. In total, analyzing a million code changes to extract feature vectors and indexing these vectors takes up to five hours. As this is a one-time effort that does not influence the response time, we consider it acceptable in practice.

3.5.3 RQ3: User Study

Study Setup

We perform a user study to measure whether DiffSearch enables users to effectively retrieve code changes within a given time budget, and to compare our approach with a regular expression-based baseline and the GitHub Search feature. To this end, we provide natural language descriptions of kinds of code changes and ask each user to find up to ten matching code changes per description within two minutes. We choose this time limit based on empirical results on code search sessions, which are reported to have a median length of 89 seconds [223], and to control the overall time participants of the study will have to spend. We then ask the users how many satisfying code changes they could find. Each user works on each kind of query with DiffSearch, the REGEX tool and GitHub Search.

Queries. The descriptions of the queries (Table 3.3) are designed with two criteria in mind. First, they cover different syntactic categories of changes, including additions (#3, #4, #7), modifications (#6), and removals (#10) of statements; changes within existing statements (#1, #2, #5, #9); and changes that surround an existing statement with a new statement (#8). Second, the queries cover a diverse range of reasons for changing code, including code improvements to increase robustness (#4, #7, #8), code cleanup (#10), changes of functionality (#6, #9), bug fixes (#1, #2, #5), and uses of a new API (#3).

Baselines. We compare DiffSearch against two existing tools that users might use to search for code changes. First, we compare against a regular expression-based approach suggested in the Stack Overflow question cited in Section 3.1, which we call REGEX. Regular expressions are well known and widely used for general search tasks. Naively applying regular expressions to the git history of many projects, as suggested on Stack Overflow, leads to unacceptably high response times (tens or even hundreds of seconds, depending on the query). Instead, we preprocess the output of *git log* by removing information unrelated to the task, such as commit messages and file names, which reduces the size of the file and makes the response time acceptable. Second, we compare against the search feature offered by GitHub, which matches free-form queries against commits, presumably through an indexing and retrieval approach applied to the commit message and the tokens involved in a commit. To ensure that our study participants search through the same dataset as DiffSearch, instead of all commits on GitHub, we create a single repository[24] with all code changes in our dataset, copied from the original version histories, and then restrict GitHub's search to this repository.

Participants and setup. We recruit ten participants, consisting of seven PhD students, two senior undergraduate students, and one senior developer. The participants do not overlap with the authors of the paper [41]. The user study is performed virtually with participants working from their offices or their homes. We ask each participant to assess for each of the three tools involved in the study their level of experience (*expert*, *advanced*, *intermediate*, or *beginner*) and their usage frequency (*weekly*,*monthly*, *yearly*, or *never used*). None of the participants has previous experience with DiffSearch. Regarding their experience with REGEX, four participants are *advanced*, five are at *intermediate* level, and one is a *beginner*. Seven participants use REGEX monthly, and three participants even weekly. For GitHub Search, one participant is *advanced*, five are *intermediate*, and four are *beginners*. Two participants use it *yearly*, four *monthly*, three *weekly*, and one has *never used* it.

The participants access DiffSearch through a web interface that resembles a standard search engine, but has two text input fields, for the old and new code, respectively.[25] For REGEX, participants use a terminal and their favorite tool to search with regular expressions, e.g., *grep*. For GitHub Search we provide a link to GitHub that already restricts the search to commits in the repository created for this user study. We provide 1,050 words of instructions to the participants, which explain the task, the query language of DiffSearch, how to search through raw diffs using REGEX, and GitHub Search.

[24]https://github.com/luca-digrazia/DatasetCommitsDiffSearch
[25]The web interface is available, see end of Section 3.1.

Table 3.3: Query descriptions for user study and summary of search results.

Id	Query description	DiffSearch / REGEX / GitHub Search										
		User 1	User 2	User 3	User 4	User 5	User 6	User 7	User 8	User 9	User 10	Total
1	*Find changes* in which a return statement that returns a literal changes to returning the result of a method call.	10/0/0	10/0/0	0/10/0	0/0/10	10/0/0	7/0/0	10/0/3	7/0/5	7/0/0	7/0/1	68/10/19
2	*Find changes* where the developer swaps the arguments of a method call.	0/0/0	0/0/0	10/0/6	10/0/1	10/0/6	0/0/0	10/0/0	10/0/0	10/0/10	10/0/0	70/0/23
3	*Find changes* that add an import of a class in the form “import somePkg.SomeClass”.	10/0/10	0/10/10	10/10/4	10/10/10	0/10/8	10/10/10	10/10/7	10/0/7	10/0/10	10/0/10	80/60/86
4	*Find changes* that add a call to close some resource, e.g., a stream or file reader.	0/0/3	10/10/1	10/10/1	10/0/10	10/10/1	0/10/0	10/10/2	10/0/10	10/0/0	10/10/1	80/60/29
5	*Find changes* where the condition of an if statement with a bodychanges from “-= null” to “!= null”.	4/0/0	10/0/0	4/5/0	0/0/10	7/0/0	0/0/1	4/0/0	4/0/0	0/0/0	5/0/0	38/7/11
6	*Find changes* that remove a method call with one argument.	10/0/10	10/1/1	10/10/10	10/0/0	10/0/10	10/10/1	10/10/0	10/0/0	10/0/6	10/0/0	100/31/38
7	*Find changes* that insert an assertion using Java’s “assert” keyword.	10/0/6	10/10/10	0/10/0	0/2/10	10/10/2	0/10/0	10/10/2	10/0/3	10/0/0	10/10/0	70/62/33
8	*Find changes* in which a code snippet is surrounded with a try/catch block.	0/0/0	0/0/5	0/0/1	0/0/0	0/0/1	10/10/3	4/10/4	10/0/3	0/0/5	1/0/5	25/0/27
9	*Find changes* where the condition of a while loop is changed.	10/0/0	10/10/1	10/2/0	10/0/0	10/0/1	0/0/1	10/0/0	0/0/0	10/0/0	10/0/0	90/13/3
10	*Find changes* that remove a call to System.out.println(...).	10/0/6	10/10/4	10/10/1	10/0/10	10/10/5	10/10/1	10/10/2	10/0/3	10/0/1	10/0/0	100/60/33
	Total	64/0/35	70/51/32	64/67/23	64/12/61	60/40/34	47/53/17	88/50/20	81/0/31	77/0/32	83/30/17	711/303/302

Quantitative Results

Table 3.3 shows the number of search results obtained using DiffSearch and REGEX. Across the entire study, the participants find 711 code changes with DiffSearch, but only 303 with REGEX and 302 with GitHub Search. Inspecting individual queries shows that, while some are harder than others, at least one user finds ten code changes for each query. For 77.0% of DiffSearch queries, users retrieve at least one code change with DiffSearch, whereas with REGEX, users get at least one code change for only 35.0% of all queries, and 60.0% of GitHub Search queries lead to at least one code change. For 65.0% of DiffSearch queries, users find the desired number of ten code changes, but only 29.0% of users succeed with REGEX and 15.0% with GitHub Search. Overall, we conclude that DiffSearch enables users to effectively find code changes, and that the approach clearly outperforms the REGEX-based and GitHub Search baseline.

Qualitative Results

To better understand the strengths and weaknesses of DiffSearch, we manually inspect queries formulated by users. All the users get enough results for query #6, e.g., with queries such as ”ID(EXPR); → _”, underlining how easy it is querying DiffSearch. Another example is query #10, where all participants use a query similar to ”System.out.println(EXPR); → _”, which yields 10 satisfying results. The user study also shows how fast the participants learn to use DiffSearch. For example, Users 2 and 5 on query #3 find zero code changes with DiffSearch, while they find 10 code changes on query #4 because they have learned more about the query syntax. As another example, User 2 for query #3 uses queries like ”_ → import LT().LT()” and ”_ → import LT<...>.LT<...>”, which are syntactically invalid. After some tries the user understands the query and they perform better on the following queries.

When asking participants about their experience after the experiment, some users report difficulties in formulating precise queries on GitHub Search. For example, for query #6 a user says: "found many other method calls with more than one argument that were removed as well". For query #7 a user states: "I could find some more code that uses assert but not specifically that inserts an assert keyword". These examples illustrate that DiffSearch is particularly useful when searching for non-trivial code changes and to avoid false positive results.

While DiffSearch clearly outperforms REGEX and GitHub Search for all ten queries, there are some user-query pairs where REGEX and GitHub Search yields more results than DiffSearch. Analyzing these cases shows two main reasons. First, some users were effective with regular expressions by searching for simple code changes that only add or only remove a single line of code. For example, for query #3, some users simply searched for "+ import (.*)". Instead, for the same query GitHub Search has the best performance because users find precise commit messages for this kind of code change. Second, some users formulated regular expression queries that are more general than the natural language description we provide and then manually filtered the results to find the ten relevant code changes. For example, for query #5, a user searched for "if((.*?))" and then manually checked for conditions that involve `null`. Finally, Users 3 and 6 find more code changes with REGEX than the other two tools. These users judge their REGEX experience with *advanced* and *intermediate*, respectively, and they both use REGEX *monthly*, which they affirm to have helped them to be effective with REGEX on this task.

We also asked for informal feedback about the three tools, to better understand their strengths and weaknesses. Users report three reasons for preferring DiffSearch over REGEX and GitHub Search. First, they find the DiffSearch query more precise than regular expression syntax or free-form queries, because it builds upon the underlying programming language. In particular, some users affirm that in two minutes they were able to type a DiffSearch query, but not a working regular expression, especially for complex queries, such as multi-line code changes. Second, REGEX often was much slower than DiffSearch because it linearly searches through all code changes, while GitHub Search often shows commits with so many hunks that it is difficult to find a specific code change. This inefficiency, especially for more complex code changes, caused some users to not find any relevant code changes in the given time. Finally, some users mention that REGEX syntax is not precise enough to formulate effective queries, leading to many false positives.

3.5.4 RQ4: Searching for Bug Fixes

As a case study for using DiffSearch, we apply it to search for instances of bug fix patterns, which could help, e.g., to establish a dataset for evaluating bug detection tools [85], automated program repair tools [10], or for training a learning-based bug detection tool [204]. We build on a set of 16 patterns defined by prior work [115], of which we use twelve (Table 3.4). The remaining four bug fix patterns are all about single-token changes, e.g., changing a numeric literal or changing a modifier, which currently cannot be expressed with our query language. For the twelve supported patterns, we formulate queries based on the descriptions of the patterns and then search for them with DiffSearch. We use two different datasets for this case study. First, a set of around 10,000 code changes, called *SStuBs commits*, that contains all those commits where the prior work [115] found instances of the bug fix patterns through custom-built analysis scripts, which we call *SStuBs*. Second, a set of around 1,000,000 code changes, called *Large*, sampled from all the repositories analyzed in the prior work.

Table 3.4 shows for each bug fix pattern how many code changes the different approaches find. DiffSearch returns a total of 15,959 code changes for the first dataset and 74,903 for the second dataset. Computing the intersection with the results retrieved by SStuBs, DiffSearch finds 79.2% of their changes, a result consistent with the Java recall computed in RQ1. Moreover, DiffSearch finds many more matching code changes, increasing the dataset from 2,867 to 15,959 examples of bug fixes. The reason is that our queries are more general than the custom analysis scripts in SStuBs and include, e.g., also code changes that perform other changes besides the specific bug fix. The number of code changes found by DiffSearch is higher than the number of commits (10k) because a single commit may match multiple patterns. For example, a change that swaps two arguments and modifies a function name will appear in patterns 5 and 8. Overall, DiffSearch is effective at finding various examples of bug fix patterns, showing the usefulness of the approach for creating large-scale datasets.

Table 3.4: Effectiveness of DiffSearch in finding instances of bug fix patterns [115].

	Description	SStuBs commits (10k)			Large (1M)
		SStuBs	DiffSearch	Both	DiffSearch
1	Change only caller	132	1,880	121	5,974
2	Change binary operator	211	347	131	2,979
3	More specific if	130	592	116	5,660
4	Less specific if	166	592	150	5,387
5	Wrong function name	1,141	1,439	935	8,109
6	Same caller, more args	557	2,108	432	11,207
7	Same caller, less args	110	2,123	75	10,798
8	Same caller, swap args	98	2,285	89	9,042
9	Change unary operator	126	134	70	6,081
10	Change binary operand	91	347	73	2,136
11	Add throws exception	60	1,834	34	3,848
12	Delete throws exception	45	2,278	44	3,682
	Total	2,867	15,959	2,270	74,903

3.5.5 RQ5: Impact of Parameters

We perform a sensitivity analysis for the two main parameters of DiffSearch: the length l of feature vectors (Section 3.3.3), and the number k of candidate matches retrieved via the feature vectors (Section 3.3.4). We select a set of values from 1,000 to 20,000 for k and from 500 to 4,000 for l, i.e., values below and above the defaults, and then measure their impact on the time to answer queries, the recall, and the size of the index.

Table 3.5 shows the results. We find that retrieving more candidate code changes, i.e., a higher k, slightly increases the response time. The reason is that matching more code changes against the query increases the time taken by the matching phase. On the positive side, increasing k increases the recall, reaching 87.3% for Java, 93.7% for Python, and 95.6% for JavaScript when k=20,000, while still providing an acceptable average response time. Parameter l increases the time to answer a query because a larger feature vector slows down the nearest neighbor search. Likewise, a larger l also increases the size of the index. Since increasing l beyond our default does not significantly increase recall, we use l=1,000 as the default to have a manageable index size and a reasonable response time. As a result, users can adjust the parameters based on their usage scenario. They can use a higher k if they prefer recall over efficiency, or a lower k if they prefer the opposite.

3.5.6 RQ6: Queries vs. Search Results

The goal of performing a search is to obtain more information than provided in the query. To assess to what extent DiffSearch serves this purpose by characterizing queries and the resulting search results in two ways. These experiments are done on all three currently supported languages, using the 80 queries described in RQ1.

First, we quantify the number of results obtained via a single query. We compute the average number of code changes retrieved by DiffSearch among the 80 queries. We find an average of 646 results for Java, 269 for Python, and 280 for JavaScript. As a result, we can conclude that typing a single DiffSearch query results in a significant amount of information retrieved.

Second, we approximate the amount of information in a query and the resulting search results by counting the number of characters they are composed of. For the results with multiple code changes we compute the average of their size. We find an average query size of 95 and an average result size of 136 for Java, an average query size of 47 and an average result size of 67 for Python, and an average query size of 34 and an average result size of 55 for JavaScript. As a result, we can conclude that the result of DiffSearch queries contains 29.9%, 29.8%, and 38.2% more information than provided in the query for Java, Python and JavaScript, respectively.

In conclusion, we show that the effort to type a DiffSearch query has benefits in the quantity of information retrieved.

Table 3.5: Impact of length l of feature vectors and number k of candidates (default configuration is (**bold**).

k	l	Response time (s)			Recall	Size of
		min	avg	max	(%)	index (GB)
Java:						
1,000	1,000	1.5	1.9	3.5	71.8	4.0
5,000	**1,000**	**1.5**	**2.2**	**9.0**	**80.7**	**4.0**
10,000	1,000	1.7	2.5	9.4	84.9	4.0
20,000	1,000	1.8	3.1	17.7	87.3	4.0
5,000	500	0.8	1.3	8.1	79.3	2.0
5,000	2,000	3.0	4.2	9.9	80.6	8.0
5,000	4,000	5.8	7.4	15.3	78.1	16.0
Python:						
1,000	1,000	3.0	4.1	5.5	81.9	4.1
5,000	**1,000**	**1.8**	**2.4**	**3.5**	**89.8**	**4.1**
10,000	1,000	3.5	5.0	8.9	91.6	4.1
20,000	1,000	4.1	6.0	12.4	93.7	4.1
5,000	500	1.0	1.6	3.1	86.6	2.0
5,000	2,000	2.7	4.9	40.8	89.8	8.1
5,000	4,000	6.1	7.9	13.1	83.4	16.3
JavaScript:						
1,000	1,000	1.2	1.9	2.8	85.4	4.0
5,000	**1,000**	**1.3**	**2.0**	**2.8**	**90.4**	**4.0**
10,000	1,000	1.4	2.3	3.3	94.0	4.0
20,000	1,000	1.8	2.9	5.7	95.6	4.0
5,000	500	0.7	1.2	2.1	90.3	2.0
5,000	2,000	3.1	4.5	5.4	92.5	8.0
5,000	4,000	5.1	9.2	12.8	88.6	16.1

3.6 Limitations and Future Work

Our approach has some limitations that will be interesting to address in future work. First, in Section 3.3.2, we explain the challenge of parsing incomplete parse trees. We extend the ANTLR4 grammar for the target programming languages with optional rules to parse incomplete snippets of code that commonly occur in hunks. These extensions cover most but not all hunks, and we plan to enable parsing of an even larger range of incomplete code snippets in the future. Second, our approach of parsing individual hunks will be non-trivial to apply to languages that make heavy use of macros, such as C. The reason is that, when trying to parse a single hunk, the definitions of macros are not available. Finally, Section 3.3.3 describes the features we design for code changes. Future work could either extend those features with other features or apply neural networks that learn to map code changes into continuous vector representations, such as CC2Vec [98] and Commit2Vec [27].

3.7 Concluding Remarks

We present a scalable and precise search engine for code changes. Given a query that describes code before and after a change, the approach retrieves within seconds relevant examples from a corpus of a million code changes. Our query language extends the underlying programming language with wildcards and placeholders, providing an intuitive way of formulating queries to search for code changes. Key to the scalability of DiffSearch is to encode both queries and code changes into a common feature space, enabling efficient retrieval of candidate search results. Matching these candidates against the query guarantees that every returned search result indeed fits the query. The approach is mostly language-agnostic, and we empirically evaluate it on Java, JavaScript, and Python. DiffSearch answers most queries in less than a second, even when searching through large datasets. The recall ranges between 80.7% and 90.4%, depending on the target language, and can be further increased at the expense of response time. We also show that users find relevant code changes more effectively with DiffSearch than with a regular expression-based search and GitHub Search. Finally, as an example of how the approach could help researchers, we use it to gather a dataset of 74,903 code changes that match recurring bug fix patterns. We envision DiffSearch to serve as a tool useful to both practitioners and researchers, and to provide a basis for future work on searching for code changes.

Chapter 4

The evolution of type annotations in Python: an empirical study

Type annotations and gradual type checkers attempt to reveal errors and facilitate maintenance in dynamically typed programming languages. Despite the availability of these features and tools, it is currently unclear how quickly developers are adopting them, what strategies they follow when doing so, and whether adding type annotations reveals more type errors. This chapter presents the first large-scale empirical study of the evolution of type annotations and type errors in Python. The study is based on an analysis of 1,414,936 type annotation changes, which we extract from 1,123,393 commits among 9,655 projects. Our results show that (i) type annotations are getting more popular, and once added, often remain unchanged in the projects for a long time, (ii) projects follow three evolution patterns for type annotation usage – *regular annotation*, *type sprints*, and *occasional uses* – and that the used pattern correlates with the number of contributors, (iii) more type annotations help find more type errors (0.704 correlation), but nevertheless, many commits (78.3%) are committed despite having such errors. Our findings show that better developer training and automated techniques for adding type annotations are needed, as most code

still remains unannotated, and they call for a better integration of gradual type checking into the development process.

4.1 Introduction

Dynamically typed languages, such as Python and JavaScript, have become the most popular languages for newly written code.[26] One reason for this popularity is their lightweight syntax, which does not require developers to specify the types of parameters, return values, fields, or variables. At the same type, the absence of static type annotations often hampers maintenance, causes type-related bugs to be missed, and limits IDE support.

The problems caused by the complete absence of type annotations has motivated optional type annotations. They offer a flexible middle ground between no type annotations at all and a fully statically typed language, enabling each developer to annotate only those types she believes to be beneficial. The two most popular dynamically typed languages, Python and JavaScript, both support optional type annotations. In particular, Python 3.5 specifies the meaning of type annotations for functions,[27] and Python 3.6 adds syntax for specifying the types of variables.[28]

In recent years, a variety of tools have been proposed for helping developers deal with type annotations in dynamically typed languages. Gradual type checkers [188, 257] use the available type annotations, possibly along with type information for popular libraries, to check for type errors. Beyond gradual type checkers, recent work proposes techniques to infer and predict type annotations, based on static analysis [7, 63, 91, 106], dynamic analysis [6, 216], probabilistic rules [278], learned predictive models [4, 92, 157, 213], and combinations of the former [203]. Such tools help annotate code with types, in particular code written before the standardized introduction of type annotations into the programming language.

Figure 4.1 shows an example of a partially type-annotated Python function and its evolution across three commits, which illustrates different kinds of type

[26]`https://octoverse.github.com/`
[27]`https://www.python.org/dev/peps/pep-0484/`
[28]`https://www.python.org/dev/peps/pep-0526/`

(a) Commit 1.

```
def f(x: float, y):
  sum = x + y
  if (sum % 2) == 0:
    return True
```

(b) Commit 2.

```
def f(x: int, y) -> bool:
  sum: int = x + y
  if (sum % 2) == 0:
    return True
```

(c) Commit 3.

```
def f(x: int, y) -> Optional[bool]:
  sum: int = x + y
  if (sum % 2) == 0:
    return True
```

Figure 4.1: Example of an evolving, partially type-annotated Python function.

annotations, how they may evolve, and type errors that may become apparent as a result. The function expects two parameters and returns whether their sum is an even number. In Commit 1, the first parameter is initially annotated to be of type `float`, whereas the second parameter remains unannotated. Commit 2 updates the parameter type of `x` from `float` to `int`, and also inserts an annotation of the return type being `bool`. Moreover, the code change inserts an annotation for the `sum` variable to be of type `int`. Type checking Commit 2 will produce a type error because not all paths through the function return a boolean, but the function implicitly returns `None` when the sum is not even. Finally, Commit 3 fixes the type error by modifying the return type annotation to `Optional[bool]`.

Several years have passed since the release of Python 3.5 in 2015 and type annotations are hypothesized to be useful to developers, but it is currently unclear how often they are adopted in practice and how this trend is evolving. How frequently do people use this feature? And how has its usage changed over time? This serves to double check if adding type information to the Python language is actually perceived as a benefit worth undertaking by developers. Moreover, the gradual type system of Python defines several standard library types such as `int`, `List`, and `Optional`. Which of those are the most useful to developers? And which of those have the most changes? We can find that adding those types is non-trivial, and they sometimes result in incorrect type annotations. What impact do the annotations have on detected type errors, and if errors are detected, do developers address them? This helps to understand if type checkers are useful during the development process. As a result, answering these and other questions consist in better understanding the adoption of type annotations in Python, identify issues that developer's commonly face in this process, and steer future research on developer tools toward the most relevant problems.

This chapter presents the first comprehensive study of the evolution of type annotations and type errors in Python. The study is performed on 9,655 of the most popular Python projects, analyzes 1,123,393 commits, and studies 668 projects in more detail, as they contain at least one annotation. We address four research questions:

- *RQ1: How does the adoption of type annotations evolve at the ecosystem level?* To better understand to what extent type annotations are getting adopted by the developer community as a whole, we study the evolution of the prevalence of type annotations across a wide range of projects.
- *RQ2: How does the usage of type annotations evolve at the project level?* This question aims at understanding the evolution of type annotations based on a commit-based timeline of single projects.
- *RQ3: How do individual type annotations evolve?* Answering this question helps understand if and how type annotations, once inserted, change over time. We also study whether type annotations are added alongside other code changes or in specific commits, and how long they remain in a code base.
- *RQ4: How do statically detectable type errors evolve and how do they relate to the type annotations in a project?* This question aims at understanding to what extent gradual type checkers can help avoid type errors, whether developers fix these errors, and how they are impacted by adding, changing, or removing annotations.

Our methodology is based on an AST-based analysis to extract type annotations, a differential analysis to understand how type annotations evolve over time, and gradual type checking on different versions of the projects.

Prior work has studied other aspects of dynamically typed programming languages, e.g., whether developers migrate from Python 2 to Python 3 [275], how linters are used in JavaScript [252], and whether type annotations in JavaScript [65] and Python [121] reveal bugs. Ore et al. assess the human effort involved in adding type annotations[187], and Hanenberg et al. study the impact of type annotations on development time [87] and maintainability [88].

Another recent study [211] is about the kinds of type annotations developers use and how the errors reported by different gradual type checkers differ. In contrast to all the above, our study focuses on the evolution of types and type errors over time, allowing us to better understand the long-term trends in adopting gradual typing.

Our study leads to several findings regarding the prevalence, characteristics, and evolution of type annotations and type errors in real-world Python code, including:

- Type annotations are getting more and more popular, but are still far from being the norm. Less than 10% of all possible code elements are currently annotated, even in projects that have at least one type annotation. This trend is slowly changing in favor of more annotations, offering a huge opportunity for researchers and practitioners to build tools that help with this process.
- Most type annotations are added alongside other code, but developers also occasionally (1.3% of all type-editing commits) work on the type annotations only.
- Developers mostly focus on annotating parameter types and return types, and less on variable types.
- Once added, many type annotations are never updated (90.1%) and many type annotations (70.4%) are still present in the latest version of a project, rewarding the effort of adding type annotations. If developers change type annotations, then optional types are commonly involved in the change.
- Most commits (78.3%) contain statically detectable type errors but are nevertheless integrated into the code base.
- Adding type annotations tends to increase the number of detected type errors (correlation of 0.704).

Our findings have several implications for developers and researchers working on developer tooling. First, we find that adding type annotations is a long-term investment because they are rarely modified, which can impact the maintainability of a code base over years. Second, the result that more than 90% of program elements are not yet annotated, both in legacy code and newly written code, is a call to arms for creating tools that infer and predict types. Recent work on learning-based type prediction is a promising step [92, 157, 203]. Third, the repetitive nature of type annotation changes pinpoints several easy to avoid mistakes, such as avoiding corner cases using a type `T` instead of its optional variant `Optional[T]`, as in our motivating example. Finally, the fact that most commits have statically detectable type errors calls for more developer awareness and better integration of gradual type checkers into the development process.

In summary, this chapter makes the following contributions:

- A comprehensive study of the evolution of type annotations and type errors in real-world Python code.
- Findings that may impact developers and teachers, as well as future work on tools and techniques for developers.
- A dataset of type annotation-related code changes to be used in future work [76], e.g., on mining and learning from these changes.

4.2 Methodology

This section discusses the methodology we use to address our research questions.

4.2.1 Extracting and Studying Type Annotations

The core concept of our study are type annotations:

Definition 4.1 (Type annotation)
A type annotation is a tuple $t_{ann} = (t, n, k, l)$, where t is a type, n is the name of an annotated program element, k is the kind of the annotated program element, and l is the code location of the type annotation.

In line with the type annotations specified by the Python language, we consider three kinds k of program elements: argument types, return types, and variable types. The code location l is specified by a file path and a line number. As an example, consider the code in Figure 5.1a. The type annotation for the annotated parameter type is (t, n, k, l), where t is `float`, n is `x`, k is "argument type", and l is line 1 of the corresponding file.

Given the type annotations in a code base, we compute the following notion of coverage, which indicates how many of all program elements that could be annotated are indeed annotated:

Definition 4.2 (Type annotation coverage)
Given a code base B, the type annotation coverage for a kind k_{target} of program elements is:

$$cov_{ann} = \frac{|\{(t,n,k,l) \in B \mid k = k_{target}\}|}{|\{\textit{All program elements of kind } k_{target}\}|}$$

For example, consider a code base that consists only of the function in Figure 5.1b. The type annotation coverage for argument types is 50%, because one out of two arguments is annotated, whereas the type annotation coverage for return types is 100%.

To extract both present and missing type annotations, we perform an AST-based static analysis of each Python file (*.py*) and each Python stub file (*.pyi*) in a version of a project. The analysis visits each node that corresponds to a possibly type-annotated program element and, if an annotation is found, extracts the corresponding tuple. We focus on type annotations in the type definition syntax added in Python 3.5 (PEP 484) and Python 3.6 (PEP 526). In contrast, we do not consider types described in informal type comments, because type comments are only partially supported by tools and because our preliminary results found type comments to occur clearly less often than annotations in proper syntax. The analysis of Python files and Python stub files is implemented on top of the LibCST and Typed AST libraries, respectively.[29]

4.2.2 Extracting and Studying Type Annotation Changes

As the primary focus of our work is to study the evolution of type annotations, we extract annotations across the history of a project and relate them to each other.

[29] `https://github.com/Instagram/LibCST` and `https://github.com/python/typed_ast`

Definition 4.3 (Type annotation change)
A type annotation change is a tuple $t_{change} = (t_{ann}^{old}, t_{ann}^{new}, c, d)$, where t_{ann}^{old} and t_{ann}^{new} are the type annotations before and after a code change, respectively, c is the kind of code change, and d is the date of the code change.

We consider three kinds of code changes: inserting, updating, and removing a type annotation. In a type annotation change that inserts or removes an annotation, the old or new annotation is undefined, respectively, which we represent with _. For example, the commits in Figure 4.1 involve four type annotation changes: an update of the argument type `x`, a newly inserted annotation for the return type of `f`, a newly inserted annotation for the variable `sum`, and an update of the return type of `f`.

To study the evolution of individual type annotations, we combine multiple type annotation changes into a history:

Definition 4.4 (Type annotation history)
A type annotation history is a sequence $[t_{change}^{1}, ..., t_{change}^{m}]$ of type annotation changes, where:

- *the t_{change}^{1} has code change kind $c =$"insert",*
- *there is at most one type annotation change with $c =$"remove" and if it exists, then it is t_{change}^{m},*
- *the kind k of program element is the same in all type annotation changes, and*
- *for consecutive type annotation changes, the new type annotation t_{ann}^{new} of the first change is the same as the old type annotation t_{ann}^{old} of the second change.*

For example, consider an extended version of the example in Figure 4.1 where the code change in the figure is preceded by a commit that adds the function without any type annotations and followed by a commit that removes the annotation of the `x` argument again. This evolution of the argument type would be represented as a type annotation history with three type annotation changes, which describe how the annotation of argument `x` gets inserted, updated, and removed, respectively.

Algorithmus 4.1 Extract type annotation changes from commits.

```
Input: Sequence $C$ of commits
Output: Set $T$ of type annotation changes (old and new type pairs)
 1: $T \leftarrow \emptyset$
 2: for commit $c$ in $C$ do
 3:     $T_{old} \leftarrow$ type annotations in code before commit $c$
 4:     $T_{new} \leftarrow$ type annotations in code after commit $c$
 5:     $d \leftarrow$ date of $c$
 6:     $T' \leftarrow \emptyset$
 7:     for $(t_{old}, t_{new}) \in T_{old} \times T_{new}$ do
 8:         if $\exists$ hunk $h$ in $c$ where $t_{old}$ in $oldLineRange(h)$
 9:         and $t_{new}$ in $newLineRange(h)$ then
10:             if $t_{old}$ and $t_{new}$ have same kind $k$
11:             and same name $n$ then
12:                 Add $(t_{old}, t_{new}, \text{“update”}, d)$ to $T'$
13:             end if
14:         end if
15:     end for
16:     $T' \leftarrow ensureSingleMatch(T')$
17:     for $t_{new}$ not yet added to $T'$ do
18:         Add $(\_, t_{new}, \text{“insert”}, d)$ to $T'$
19:     end for
20:     for $t_{old}$ not yet added to $T'$ do
21:         Add $(t_{old}, \_, \text{“remove”}, d)$ to $T'$
22:     end for
23:     $T \leftarrow T \cup T'$
24: end for
```

We compute type annotation changes and type annotation histories by combining the annotations extracted by our AST-based analysis across the commit history of a repository. Tracking annotations across histories is a non-trivial challenge, e.g., because line numbers change due to removed and added code, or because developers may modify multiple type annotations in a single commit.

Algorithm 4.1 summarizes our approach for addressing this challenge. The algorithm iterates through a sequence of commits and extracts a set of type annotation changes from it. At first, lines 3 and 4 extract the type annotations from the old and the new version of a commit, respectively. Then, the algorithm compares these annotations based on their code location, the kind of the annotated element, and the name of the annotated element. The goal is to find matches, i.e., pairs of an existing annotation and a revised version of it, and that hence, should be combined into a type annotation update. To this end, the algorithm builds upon the concept of hunks, i.e., consecutive lines that are changed together. Concretely, lines 8 to 16 check whether a pair of an old and a new type annotation fit into the line range of a hunk in the commit, and if so, compares the kind and name of the annotated program element. Because we collect the life of type annotations from insertion to removing (if removed), the *ensureSingleMatch* function checks if it is an update of a program element already collected or if it is a different program element without creating duplicate elements. After finding pairs of type annotations that are changed in the commit, lines 17 to 22 consider annotations that exist only in the old or only in the new version of the commit. These annotations are added to the set of type annotation changes as "inserted" and "removed" annotations, respectively.

Because Algorithm 4.1 is heuristic, we validate the accuracy of the type annotation changes it extracts by manually inspecting 85 histories with a total of 204 type annotation code changes. We randomly sample these type annotation histories based on three categories: (i) annotations that are never updated and still present in the last analyzed version of the project, (ii) annotations that are never updated but removed at some point during the commit history, (iii) annotations that are updated multiple times. For each sampled history, we carefully inspect the commits involving the annotation and establish a ground truth history. We find that 90.1% of the automatically extracted histories match the manually established ground truth, i.e., the vast majority of histories is correctly extracted. The main reasons for (partially) incorrect type annotation histories are mismatched annotations due to multiple identifiers with the same name in the same file, and renamed or deleted files that our analysis does not track.

4.2.3 Gathering and Studying of Type Errors

We study the evolution of type errors by running a gradual type checker on different commits in the history of a project. There are several popular type checkers for Python, e.g., pyre, mypy, and pytype. For our study, we focus on pyre, because it is industrially used, e.g., at Facebook, and could successfully analyze the studied projects.

The kinds of type errors reported by pyre and other gradual type checkers fall into two categories. One category of errors are those caused by missing dependencies, e.g., when the type checker cannot find an imported class or cannot resolve a reference to a type. These errors are unlikely to occur when a type checker is used by the project developers, assuming that the developers create a proper configuration that resolves all external dependencies. In contrast, eliminating these errors in a large-scale study is difficult because resolving all dependencies and configuring the type checker to find the dependencies is non-trivial. The second category of errors are the actual type errors, which result from inconsistencies between inferred and annotated types of values and the uses of these values. For example, these errors occur because a function argument is incompatible with the declared parameter type or because a method overrides another method with an incompatible type signature. We focus our study on the second category of errors, and unless otherwise mentioned, ignore the first category, providing a realistic view of what errors the developers of a project would see when using a type checker.

4.2.4 Selection of Projects to Study

As subjects for our study we select a wide range of open-source projects based on their creation time and their popularity. At first, we gather the list of all Python projects at GitHub via the GitHub API.[30] We group the projects by their creation date, considering projects created in the years 2010 to 2019, into ten groups that each cover one year. Then, we sort the projects in each group by their number of stars and select the top-1000 per group, which yields a total of 10,000 projects to study. The rationale for first grouping and then sampling is to avoid biasing our study toward projects created in a particular time frame, e.g., mostly old projects. Removing projects that we could not clone, e.g., because they became unavailable since the beginning of our study, the total number of analyzed repositories is 9,655.

[30] `https://api.github.com/search/`

4.3 Results

This section presents the results we obtain when addressing our four research questions. Before going through the research questions, we give an overview of the analyzed data. In total, the study involves 1,123,393 commits in 9,655 repositories. Our analysis extracts 1,414,936 type annotation changes from these commits. These type annotation changes correspond to 61,861 commits and 668 repositories that have at least one type annotation change. Our results are for these 668 projects. As general statistics, the number of commits with at least one type annotation grows every year. An early adoption started already in 2015, where 3.8% of the commits contain at least one type annotation and this number of commits grows every year until reaching 10.9% in 2021.

4.3.1 RQ1: Ecosystem-level Evolution of Type Annotations

To understand whether type annotations are becoming more common in the Python ecosystem as a whole, we analyze the evolution across all studied projects. The goal is to understand trends in the ecosystem, e.g., caused by the introduction of new programming language features or tools. We perform this analysis from two points of view. First, we analyze the evolution of the absolute number of type annotations. Second, we measure the evolution of type annotation coverage.

How is the total number of type annotations evolving?

We measure the overall number of type annotations and the overall number of lines of code between 2015 and 2021. Figure 4.2 shows the results, taking a snapshot of each project on October 1 of each year. The main observation is that both the absolute number of type annotations and the number of type annotations per line of code are steadily increasing in a roughly linear manner since 2017. In 2021, there are around 50.1 annotations per 1,000 lines of code.

To better understand the relative importance of annotations provided in regular Python files (*.py*) and Python stub files (*.pyi*), Figure 4.2 distinguishes between them. It shows that the vast majority, e.g., 99.1% of all annotations present in 2021, are provided in regular Python files. A manual inspection of 20 stub files sampled from 13 projects shows that most annotations provided in

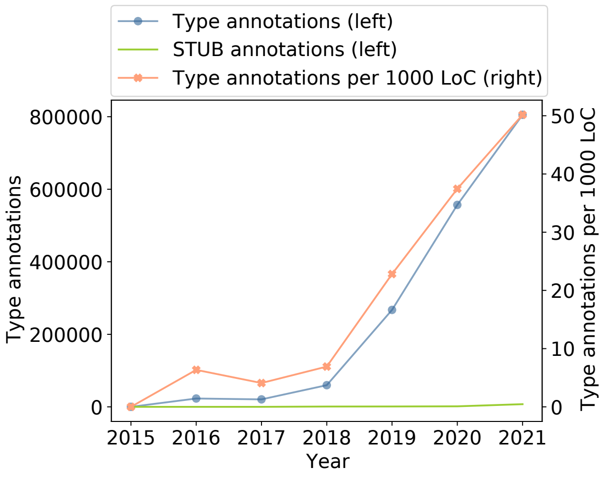


Figure 4.2: Evolution of type annotations across all projects.

stub files are about APIs of external libraries (17 out of the 20 files), for example, native libraries accessed via Python's native bindings. Two of the remaining three files are automatically generated. Given the relatively low number of annotations in stub files and their focus on external libraries, the remainder of the study considers only annotations in regular Python files.

Summary: Type annotations are getting more and more popular, with an increase of about 15 type annotation per 1,000 line of code after 2017 and reaching 50.1 annotations per 1,000 lines of code in 2021.

How is the type annotation coverage evolving?

This question is important to understand how much of the available "annotation potential" developers are currently using. Out of all 9,655 projects we study, only 668 (7%) use type annotations at all. That is, six years after the introduction of type annotations into the language, the large majority of projects is not yet using this feature. Figure 4.3 takes a detailed look into those 668 projects that use

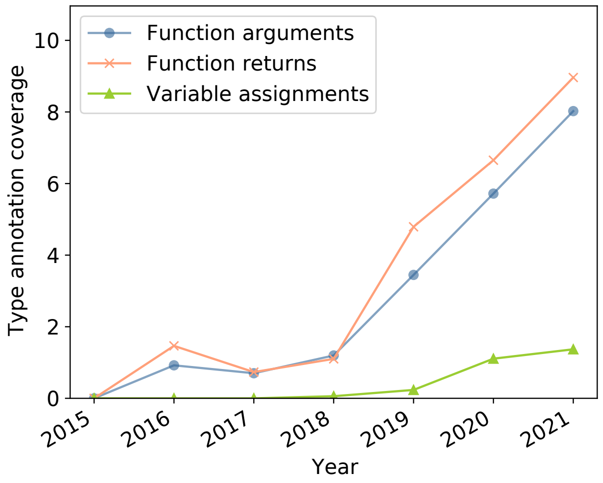


Figure 4.3: Evolution of program elements with and without type annotations.

type annotations. The figure shows the type annotation coverage for function arguments, return values, and variables on October 1 of each year. The results allow for several observations. First, the type annotation coverage is steadily increasing. Second, developers prefer to annotate function argument types and return types, but focus less on variable types. Third, despite the clear upward trend, the type annotation coverage is still relatively low, with an average of around 8% for function arguments and return values.

To put the type annotation coverage in perspective, we consider a project known for its heavy use of type annotations: *mypy*[31], which is one of the gradual type checkers for Python. This project has a type annotation coverage of 62.2% for parameter types, 94.9% for return types, and 23.4% for variable types. A manual inspection shows two main reasons for leaving program elements unannotated. First, the developers do not annotate `self` parameters, as `self` always has the type of the current class, and hence, does not really need a type annotation. Second, a significant number of unannotated local variables

[31]https://github.com/python/mypy

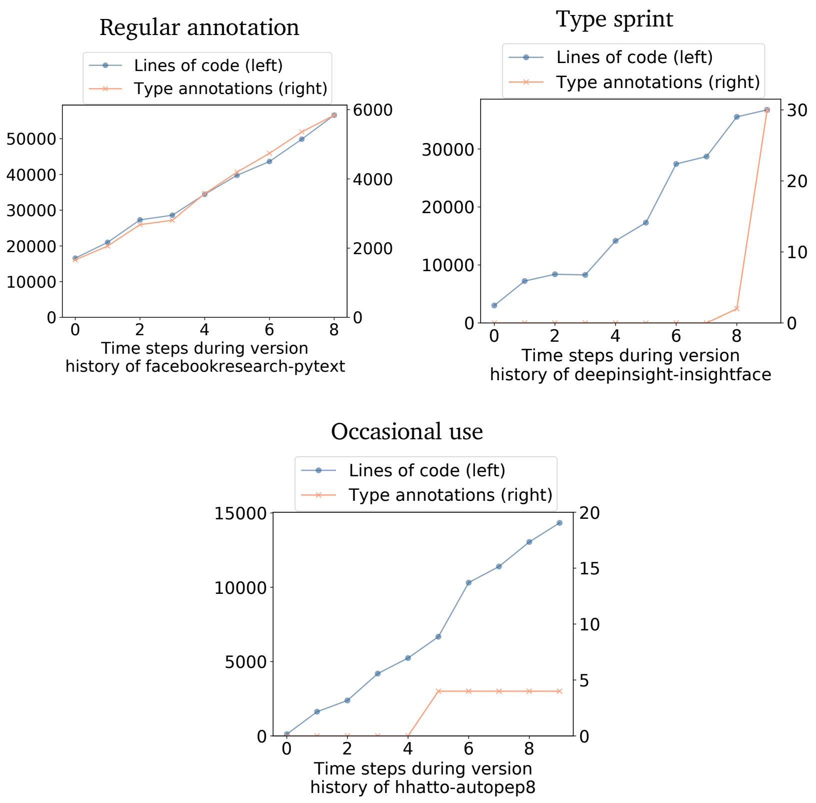


Figure 4.4: Per-project evolution of three representative projects.

have types that can be easily inferred by a gradual type checker, e.g., because a variable is assigned the result of a constructor call or the variable is assigned an annotated variable. Omitting such annotations fits the philosophy behind gradual typing, i.e., to annotate types when it is helpful without cluttering the code with unnecessary annotations.

Summary: Type annotations are not yet the norm, with less than 10% of all possible code elements being currently annotated, but there is a clear upward trend. Function arguments and return types are annotated more commonly than variables.

4.3.2 RQ2: Project-level Evolution of Type Annotations

After considering the Python ecosystem as a whole in RQ1, we now study the evolution of type annotations within individual projects. To this end, we measure how many type annotations a project has at different points during its lifetime, where lifetime means all commits from creating the project until the end of our measurement period (end of 2021). Putting the absolute number of type annotations in perspective, we also measure the number of lines of code at each point during the project lifetime. A commonality of almost all studied projects is that the number of type annotations is rarely decreasing, but instead grows continuously, i.e., once annotations are added, developers rarely remove them.

By inspecting the evolution of type annotations of various projects, we identify three common evolution patterns, illustrated in Figure 4.4 with three representative projects. For each project, the plot shows how the code size and the number of type annotations have evolved throughout the project's history.

- *Regular annotation*. Some projects, such as facebookresearch-pytext,[32] have adopted type annotations throughout their entire history and regularly add annotations as the project is growing. The typical evolution pattern of these projects is that the number of type annotations is growing at roughly the same rate as the overall code size. As can be observed by comparing the absolute numbers of lines of code (left axis in Figure 4.4) and type annotations (right axis), such projects often have significantly more type annotations than the average project. For the specific example, there are about 100 annotations per 1,000 lines of code, whereas the average project reaches, even in 2021, only about 30 annotations per 1,000 lines of code (Figure 4.2).

[32]https://github.com/facebookresearch/pytext

Algorithmus 4.2 Determine project-level evolution pattern.

```
Input: Project $P$
Output: Evolution pattern of $P$
        Divide $P$ into 10 time steps of equal number of commits
 1: $P.annotations \leftarrow$ Number of annotations for each time step
 2: $P.slope \leftarrow$ Annotation evolution slope for each time step
 3: if $max(P.annotations) < 15$ or $P.slope.count(0.0) > 8$ then
 4:     return "Occasional use"
 5: else if $P.slope.count(0.0) >= 4$ then
 6:     return "Type sprints"
 7: else if $Average(P.slope) >= 0$ then
 8:     return "Regular annotators"
 9: else
10:     return "Other"
11: end if
```

- *Type sprints*. Some other projects, e.g., deepinsight-insightface,[33] have invested into type annotations during a focused, sprint-like effort, where many annotations are added at once, but otherwise do not regularly add annotations. A variant of this pattern is a step-like curve of the number of type annotations, i.e., projects that add annotations in multiple yet non-continuous efforts.
- *Occasional use*. Some projects, such as hhatto-autopep8,[34], have only a small number of annotations, typically added in a single or very few files. This kind of project is included into the study because we consider all projects with at least one type annotation.

To measure the prevalence of these three patterns across all studied projects, Algorithm 4.2 heuristically determines whether a project fits any of the patterns. The algorithm divides the commit history of a project into ten equally sized steps, and it then checks the number of annotations present at each step and the slope from the previous to the current time step. For example, if the average slope across all time steps is positive, then the algorithm classifies the project as "Regular annotation", whereas a project with four or more time steps that do not

[33] https://github.com/deepinsight/insightface
[34] https://github.com/hhatto/autopep8

add any annotation is classified as "Occasional use". In the algorithm, $P.slope$ refers to the list of slopes observed at different time steps, and $P.slope(0.0)$ checks how many of these slope values are equal to zero. We validate Algorithm 4.2 in three steps. First, we generate the evolution plots of all studied projects. Second, we manually inspected 35 randomly selected plots and, looking at the curve, manually label each of them. Third, we run Algorithm 4.2 and compare the labels produced by the algorithm with our manual labels. During this validation, the algorithmically produced labels all match our manual labeling.

Running the algorithm across all 668 projects shows that 44.4% perform "Regular annotation", 28.1% use "Type sprints", and 25.4% are "Occasional use". The remaining 2.1% are "Other", i.e., their evolution does not fit any of the three patterns.

We also study the relation between the patterns and the characteristics of the project, such as the number of stars and contributors. While most characteristics are independent of a project's evolution pattern, we find a relationship with the number of contributors. "Regular annotation" projects have an average of 62 contributors, projects using "Type sprints" have 45 contributors, and projects with "occasional use" have only 25 contributors, on average. These numbers show that regularly adding type annotations is practice followed particularly in large repositories with a more solid organization, presumably because type annotations help coordinate between a large number of developers.

Summary: Most projects follow one of three evolution patterns when adding type annotations: "regular annotation", "type sprints", and "occasional use". Projects with more contributors tend to use "regular annotation", whereas projects with few contributors tend to follow "occasional use".

4.3.3 RQ3: Evolution of Individual Type Annotations

The following studies how individual type annotations evolve, which allows us to better understand how developers insert, modify, and remove type annotations.

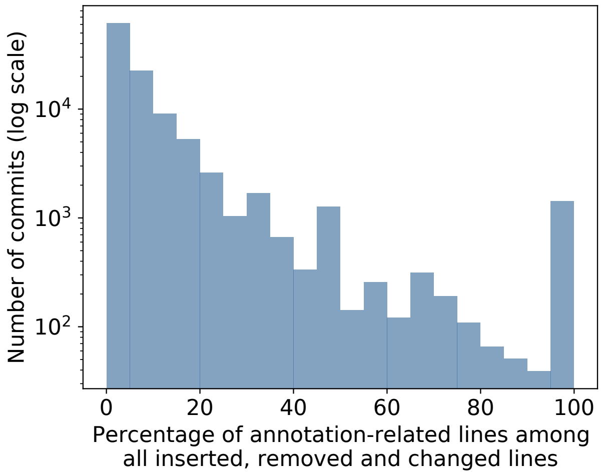


Figure 4.5: Percentage of annotation-related, edited lines among all edited lines.

When do developers edit types?

Since type annotations are optional in Python, developers can freely choose when to add or edit them. In particular, a developer can add new type annotations along with other code, e.g., along with a newly added function, or in a separate step later on, e.g., as part of a code improvement session. To understand when developers insert or modify types in a code base, we analyze all commits in the dataset that affect at least one type annotation. For each such commit, we compute how many of all lines edit a type annotation. The resulting value is a percentage, where 100% means that the commit is only to edit type annotations, and a value closer to 0% means that more other code is edited alongside the type annotation edit.

The results are shown in Figure 4.5. We see a bi-modal distribution, where the majority of commits edit a significant number of other lines in addition to editing type annotations. At the same time, there are a non-negligible number of commits that exclusively edit type annotations, showing that developers at least sometimes specifically focus on editing type annotations.

(a) Commit 1.

```
def pdist2(X: torch.Tensor,
  Z: torch.Tensor = None,
  order: PDist2Order =
        PDist2Order.d_second)
  -> torch.Tensor:
```

(b) Commit 2.

```
def pdist2( X,
  Z = None,
  order = PDist2Order.d_second):
# type: (torch.Tensor, torch.Tensor,
# PDist2Order) -> torch.Tensor
```

Figure 4.6: Example of removing a type annotation.

Summary: Most type annotations are edited alongside other code, but developers also occasionally (1.3% of all type-editing commits) work on the type annotations only.

How long do type annotations remain in a code base?

Answering this question helps understand whether adding type annotations is a long-term investment. We address the question in two ways. At first, we study how many of all ever added type annotations are still present in the latest version of the projects. To this end, we compute for each repository the number of type annotation changes that insert an annotation. In addition, we analyze the latest version of each repository, cloned on March 7, 2022, and compute how many type annotations it contains. In absolute values, 70.4% of all annotations "survive" until the latest version of a repository. For some projects, shown in the upper-right corner of the figure, all ever added annotations are still present in the latest version.

Second, we consider all type annotation histories in the dataset where the last change is a commit that removes the annotation. We compute the lifetime of each such annotation as the difference between the first and the last date in the history. In total, we find that 29.6% of all type annotations eventually get removed. We analyze in detail the removed type annotations. Their average lifetime is 160 days, showing that even annotations that get removed remain in the code for a while. An example is shown in Figure 4.6, where the types are removed from the source code and saved in a comment.[35] The commit messages says that the developers are adding support for Python 2.7, which does not support the type annotation syntax yet. We inspect a random sample of 25 of all removed annotations and classify them into three categories. We find that in 48% of the cases the entire files are removed or renamed, in 40% of the cases the program element is removed or renamed, and in 12% of cases types are explicitly removed, e.g., for supporting Python 2 or to simplify the code as shown in Figure 4.6. As a result, we can affirm that type annotations are a long-term investment because only in very few cases types are explicitly removed again.

Summary: 70.4% of all ever inserted type annotations are still present in the latest version of a repository, and those type annotations that get removed at some point "live" for an average of 126 days.

(How) do type annotations change?

Once type annotations are added, developers may modify them, e.g., to fix a wrong annotation or because the annotated code is evolving. In this research question we do not consider types that are removed. We study type annotation changes by, at first, investigating how often type annotations are updated at all. To this end, we analyze all extracted type annotation histories and compute how many updates of a type annotation they contain.

Figure 4.7 shows how many type annotations we find that are updated a specific number of times. The plot does *not* show the vast majority (90.1%) of all type annotations that are never updated. Overall, we count 139,586 annotation updates, with an average of 1.8 updates for each type annotation that gets updated at least once. The maximum number of observed updates is 25, which is an outlier though. Out of those annotations that get updated at all, most are updated five times or less. Figure 4.8 shows an example. In this case the type annotation is updated to `Sequence`,[36], and later to the user type `ModelField`.[37]

[35]https://github.com/erikwijmans/Pointnet2_PyTorch/commit/a89c4d1
[36]https://github.com/tiangolo/fastapi/commit/c20c9d8
[37]https://github.com/tiangolo/fastapi/commit/f7b7ed0

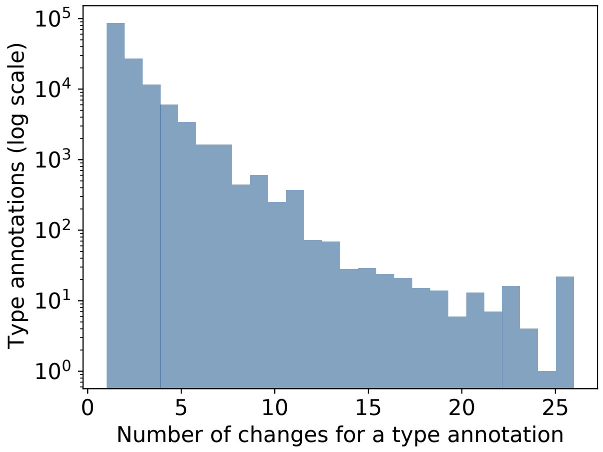


Figure 4.7: Number of times that the same type annotation is updated by developers. Not shown are the 90.1% of all type annotations with zero updates.

To better understand how annotations that get updated evolve, we analyze which kind of type annotation updates are most common. Figure 4.9 shows the results of this analysis, where the three plots show the five most commonly observed updates for argument types, return types, and variable types, respectively. We show all types that are part of the Python language and its standard library as-is, e.g., `str` and `Optional[int]`, and abstract all user-defined types into `UserType`. The results allow for two observations. First, many type annotation updates involve custom types. Second, many updates affect optional types, e.g., changing `str` to `Optional[str]`. In total, we count 14,750 type annotation updates involving optional types. A manual inspection of some of these updates shows that developers easily get confused about whether a parameter is optional or whether a variable should be immediately initialized. Figure 4.10 shows an example, where the code on the left is type-incorrect, which the developer then fixed.[38]

[38]https://github.com/gogcom/galaxy-integrations-python-api/commit/18f6cd7

(a) Commit 1.

```
def request_params_to_args(
  required_params: List[Field], ...
) -> Tuple[Dict[str, Any], List[ErrorWrapper]]:
```

(b) Commit 2.

```
def request_params_to_args(
  required_params: Sequence[Field], ...
) -> Tuple[Dict[str, Any], List[ErrorWrapper]]:
```

(c) Commit 3.

```
def request_params_to_args(
  required_params: Sequence[ModelField], ...
) -> Tuple[Dict[str, Any], List[ErrorWrapper]]:
```

Figure 4.8: Example of a type annotation updated multiple times.

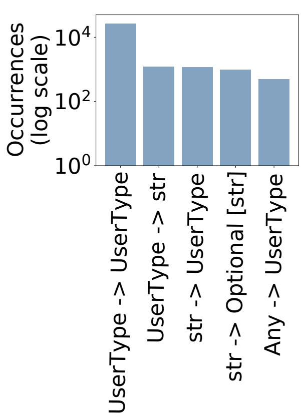
Summary: Most type annotations (90.1%) never get updated. For those that get updated, a frequent update pattern involves custom and optional types, which seems a common source of confusion.

4.3.4 RQ4: Type Errors vs. Type Annotations

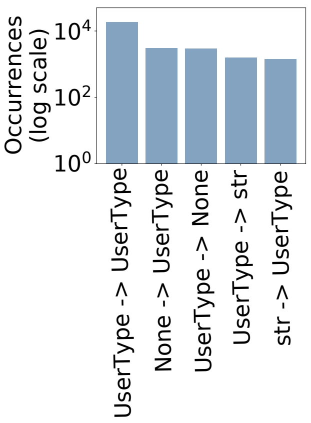
In this last research question, we inspect the number of type errors and their relationship with type annotations. We divide this analysis into three parts. First, we compute how many type errors are in these repositories. Second, we check if there is a correlation between the number of type errors and type annotations. Third, we analyze if insertions of type annotations increase the number of type errors. These three parts are performed on all the 668 projects that have at least one annotation.

How common are type errors?

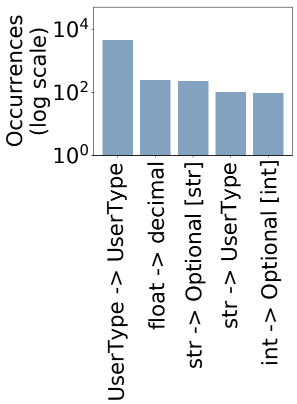
This research question is important to understand what value gradual type checkers could add to real-world Python projects and whether today's developers



Type changes in function arguments.



Type changes in function return.



Type changes in variable assignment.

Figure 4.9: Most common kinds of type annotation changes.

(a) Commit 1.

```
class LicenseInfo():
  license_type: str
  owner: str = None
```

(b) Commit 2.

```
class LicenseInfo():
  license_type: str
  owner: Optional[str] = None
```

Figure 4.10: Example of adding a wrong type annotation and then updating it with *Optional* type.

are using these tools. For each analyzed project, we run the type checker on each commit in the project's history and then count the number of non-dependency-related type errors (Section 4.2.3). We find that 78.3% of the analyzed snapshots have at least one type error. On average, there are 6 type errors per 1000 lines of code. This result indicates that type checking is not yet part of the typical development routine, calling for better tool support and more developer awareness.

To better understand the kinds of detected type errors, we analyze what kinds of errors are most common in the most recent versions of the studied projects. We find a total of 90,871 type errors and that a few kinds of errors occur repeatedly, in particular the error *incompatible variable types* (17.6%) and *incompatible parameter types* (12.6%).[39]

Summary: Most projects have statically detectable type errors, and type errors seem to not prevent developers from committing code. A few kinds of mistakes account for most type errors.

How does the number of type errors depend on the number of type annotations?

One major goal of introducing type annotations is to statically detect otherwise missed type errors. The reason is that most gradual type checkers, including the pyre tool used here, run additional type checks if more annotations are present. Even if these tools are not perfect, several studies proved the usefulness

[39] https://pyre-check.org/docs/errors/#error-codes

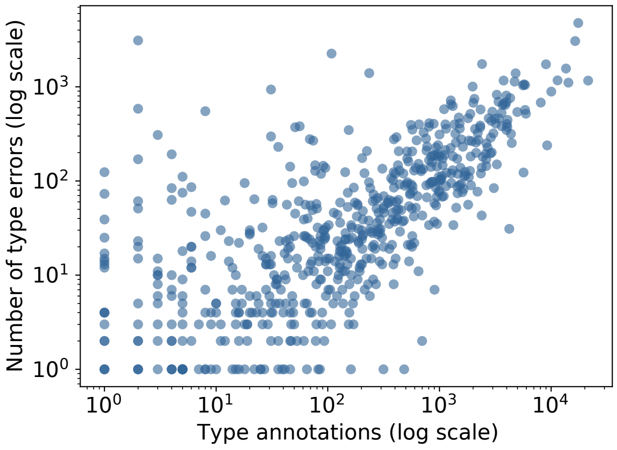


Figure 4.11: Relation between type errors and type annotations in a project (correlation: 0.704).

of type checkers [65, 121], so we decide to use pyre for this research question. To check if type annotations indeed provide this benefit in practice, we compute the correlation between the number of type errors and the number of type annotations in each project. Figure 4.11 visualizes the relation between these two measures, showing that there is a significant correlation (Pearson coefficient of 0.704) between them. We conclude that adding type annotations is only the first step toward improving type correctness, and the developers also need to introduce type checking into their developing routine.

Summary: Adding type annotations positively correlates with an increase in the number of detected type errors (correlation: 0.704). Developers should introduce type checks in their developing routine to find and fix such errors early on.

How does the number of type errors evolve when type annotations evolve?

The following aims to understand how evolving the type annotations of a project impacts the statically detectable type errors. For this purpose, we extract from all type annotation changes only those where all lines changed in the commit correspond to only adding type annotations, which we call *pure commits*. Pure commits are interesting because they allow us to study in isolation the effect on the code base of adding type annotations.

For each pure commit, we compare the number of type errors before and after the type annotation change. We find that 81 commits introduce more errors, 34 commits reduce the type errors and 319 commits keep the same number of errors. While in most cases (319) the number of type errors remains the same, the kind of pure commit has a significant impact on incrementing the number of errors.

Summary: Adding type annotations can introduce new type errors, so this process should come with the usage of type checkers.

4.4 Discussion

Implications for developers and project managers. The overall trend is that type annotations are increasingly popular (RQ 1), suggesting that developers should pick up the habit of adding annotations as they write code. Regularly adding annotations is common especially for projects with many contributors (RQ 2). Adding to benefits reported by others [65, 87, 121], our results provide empirical motivation for adding type annotations, such as the fact that more annotations help find more type errors (RQ 4) and the observation that most annotations remain in the code for a long time (RQ 3). We also pinpoint specific update patterns for individual annotations, which could help developers to avoid recurring mistakes, e.g., related to `optional` types (RQ 3). Finally, we show that developers do not need to annotate every program, because even in projects with heavy use of annotations, self-explanatory parameters and variables often remain unannotated (RQ 1).

Implications for researchers and tool builders. Even though type annotations are being used more and more, the large majority of code elements that could be annotated currently remains unannotated (RQ 1). While probably not all code in all projects needs type annotations, we see a huge potential for techniques that automate the process of adding types into an existing code base, such as neural type prediction models [4, 92, 157, 203]. Another promising direction is to improve the integration of type checking into the development process. The fact that many commits contain type errors found by a type checker (RQ 4), but nevertheless are committed, shows that type checking currently is not yet standard. Better understanding the reasons for this phenomenon will be interesting future work.

Threats to validity. Our selection of projects is based on popularity and the projects' creation time. Another selection strategy, e.g., based on application domains, might give other results. We focus on popular projects because they overall have a higher impact and are more likely to represent serious development efforts than, e.g., small toy projects or student assignments. To study type errors, we use a single type checker, pyre, and other type checkers may give other type errors. See Rak-amnouykit et al. [211] for a discussion of the subtle differences between the type systems behind pyre and mypy. Some of our results are based on manual inspection and heuristic algorithms, which likely are imperfect. To mitigate this threat, we carefully check all results and make them available as a reference for future work. Finally, our study focuses on a single language, Python, and we cannot claim that our results will generalize to others. Comparing the evolution of type annotations across different languages will be interesting future work.

4.5 Concluding Remarks

This chapter presents a large-scale empirical study of the characteristics and evolution of type annotations and type errors in Python. Our methodology statically analyzes individual commits of projects, extracts type annotations, combines them into histories that show the evolution of the annotations, and type checks different commits of projects. We extract 1.4 million type annotation changes from 9,655 repositories. Our results show that type annotations are clearly gaining traction, yet the large majority of code elements that could be annotated currently remains unannotated. While probably not all code in all projects needs type annotations, we see a huge potential for techniques that automate the process of adding types into an existing code base, such as neural type prediction models [4, 92, 157, 203]. Finally, many developers seem to not regularly check their code for statically detectable type errors, or if they do, commit the code despite such errors. We recommend to increase developer awareness and to better integrate gradual type checkers into the development process to alleviate this situation.

Chapter 5

PyTy: Repairing Static Type Errors in Python

Gradual typing enables developers to annotate types of their own choosing, offering a flexible middle ground between no type annotations and a fully statically typed language. As more and more code bases get type-annotated, static type checkers detect an increasingly large number of type errors. Unfortunately, fixing these errors requires manual effort, hampering the adoption of gradual typing in practice. This chapter presents PyTy, an automated program repair approach targeted at statically detectable type errors in Python. The problem of repairing type errors deserves specific attention because it exposes particular repair patterns, offers a warning message with hints about where and how to apply a fix, and because gradual type checking serves as an automatic way to validate fixes. We addresses this problem through three contributions: (i) an empirical study that investigates how developers fix Python type errors, showing a diverse set of fixing strategies with some recurring patterns; (ii) an approach to automatically extract type error fixes, which enables us to create a dataset of 2,766 error-fix pairs from 176 GitHub repositories, named PyTyDefects; (iii) the first learning-based repair technique for fixing type errors in Python.

Motivated by the relative data scarcity of the problem, the neural model at the core of PyTy is trained via cross-lingual transfer learning. Our evaluation shows that PyTy offers fixes for ten frequent categories of type errors, successfully addressing 85.4% of 281 real-world errors. This effectiveness outperforms state-of-the-art large language models asked to repair type errors (by 2.1x) and complements a previous technique aimed at type errors that manifest at runtime. Finally, 20 out of 30 pull requests with PyTy-suggested fixes have been merged by developers, showing the usefulness of PyTy in practice.

5.1 Introduction

Dynamically typed languages, such as Python and JavaScript, have become very popular.[40] One reason is their lightweight syntax, which does not require developers to specify types for parameters, return values, or variables. Because this flexibility may negatively affect the maintainability and robustness of code, in 2015, Python adopted optional type annotations, enabling developers to annotate types of their choosing.

5.1.1 Context

Since their introduction into the Python language, type annotations have been getting increasingly popular [43]. To support developers, several automated approaches for adding type annotations to existing code bases have been proposed, e.g., TypeWriter [203], DeepTyper [92], Typilus [4], and work by Xu et al. [278]. While adding type annotations is generally considered a step forward, newly added annotations often reveal previously unnoticed type errors, which can be easily detected with a static type checker. Unfortunately, developers commonly lack the time to fix these errors [43], which hampers the usefulness of gradual typing.

Figure 5.1 shows two real-world, statically detectable type errors along with their fixes, as performed by developers. The error presented in Figure 5.1a is caused by passing the arguments to a function in the wrong order [204],

[40] https://octoverse.github.com/#top-languages-over-the-years

```
def draw_texture_rectangle(
  texture: Texture,
  scale: float = 1):
...
draw_text_rectangle(scale, texture)
```

(a) Code with a type error.

```
def draw_texture_rectangle(
  texture: Texture,
  scale: float = 1):
...
draw_text_rectangle(texture, scale)
```

(b) Type error fixed by swapping arguments.

```
def _decorate_async_function(
  method: Callable,
  method_name: str = None):
```

(c) Code with a type error.

```
def _decorate_async_function(
  method: Callable,
  method_name: Optional[str] = None):
```

(d) Type error fixed by adding an `Optional` annotation.

Figure 5.1: Examples of type errors fixed by PyTy.

i.e., a kind of problem that in statically typed languages often can be prevented by the type system. The developers fix the problem by swapping the arguments.[41] The error presented in Figure 5.1c is caused by annotating a parameter to be a string, while at the same time, initializing it to `None`, which is type-incompatible with `str`. To fix this error, the developer modifies the type annotation to `Optional[str]`.[42] As illustrated by these examples, there may be many ways of addressing different type errors in Python, and finding the right fix for a given error is non-trivial.

5.1.2 Significance

Organizations with large Python code bases invest significant efforts toward using type annotations and type checkers. For example, Google's Python style guide mentions that developers are "strongly encouraged to enable Python type analysis" because "The type checker will convert many runtime errors to build-time errors".[43] Likewise, Dropbox type-checked over four million lines of Python in 2019, because "A type checker will find many subtle (and not so subtle) bugs. A typical example is forgetting to handle a *None* value or some other special

[41] https://github.com/pythonarcade/arcade/commit/c6aL883
[42] https://github.com/awslabs/aws-lambda-powertools-python/3898e55
[43] https://google.github.io/styleguide/pyguide.html#2212-pros

condition".[44] Finally, Meta "use[s] it extensively to maintain the codebases of Facebook and Instagram".[45]

To handle type errors in legacy code and type errors revealed by adding type annotations to previously unannotated code, an automated technique to help developers fix such errors would be desirable. However, despite the increasing popularity of automated program repair (APR) [134], there currently is no APR approach targeting static type errors in Python. Compared to repair scenarios targeted by existing APR approaches, fixing type errors in Python differs in three important ways, making the problem particularly amenable to automated repair. First, type errors require specific fix patterns, which an approach specifically targeting such errors can exploit. Second, when a gradual type checker reports a type error, the report includes an error message that may offer hints about the location and nature of the problem. Third, the gradual type checker also offers an automatic oracle, which an APR technique can use to validate candidate fixes.

5.1.3 Approach

This chapter introduces PyTy, the first APR approach for static type errors in Python. To guide the design of PyTy, we investigate in a preliminary study how developers typically fix type errors. The study investigates (i) how repetitive type errors and their fixes are, (ii) how difficult it is to localize the fix location, and (iii) to what extent the error message provided by a type checker helps in finding the fix. In short, the results show that there are recurring fix patterns, but ambiguous rules for when to apply them, and that the locations and error messages provided by the type checker are valuable information.

Based on the results of the preliminary study, we design PyTy as a data-driven approach. This kind of approach requires a dataset for training and evaluation. However, automatically collecting a large-scale dataset of type error fixes is challenging because it requires identifying relevant commits and isolating the type error fixes in these commits. We address these challenges through an automated approach that combines gradual type checking and delta debugging [284]. Using this approach, we obtain 2,766 real-world pairs of type

[44] https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python
[45] https://developers.facebook.com/blog/post/2021/05/10/eli5-pyre-fast-error-flagging-python-codebases/

errors and corresponding single-hunk fixes from 176 GitHub repositories. To the best of our knowledge, our PyTyDefects dataset is the first of its kind.

The core of PyTy is a neural type error repair model. Motivated by the relative data scarcity of the problem, we present a cross-lingual transfer learning approach. Specifically, we base PyTy on the existing APR system TFix [19], which has been trained to fix linter warnings in JavaScript code. By fine-tuning the TFix model with PyTyDefects, we retain the knowledge learned from fixing JavaScript code and apply it to fixing Python type errors. To ensure that every fix suggested by PyTy indeed fixes the targeted type error, the approach checks candidate fixes with a gradual type checker, and returns a fix only if it removes the error.

5.1.4 Results

Our evaluation on a held-out subset of 281 type error fixes shows that PyTy finds a fix that removes type errors for 85.4% of all errors. Moreover, 54.4% of the predicted fixes exactly match the developer's fix. Comparing PyTy with previous work, we find that it clearly outperforms several state-of-the-art large language models (text-davinci-003, gpt-3.5-turbo, and gpt-4) asked to repair type errors (54.4% vs. 26.4% exact matches) and complements a technique aimed at type errors that manifest at runtime [186]. As evidence of the usefulness of PyTy in practice, 20 out of 30 GitHub pull requests with PyTy-suggested fixes have been merged by the developers. Finally, we also validate the automatically gathered PyTyDefects dataset underlying our approach, and find that almost all gathered fixes are minimal and correct.

5.1.5 Contributions

In summary, the contributions of this chapter are:

- An empirical study of how developers fix type errors.
- A technique to extract type errors and corresponding fixes through a combination of gradual type checking and delta debugging, which yields the first dataset of its kind, with 2,766 type error-fix pairs from 176 GitHub repositories.

- Cross-language transfer learning that uses a model pre-trained on JavaScript to repair type errors in Python.
- Empirical evidence of the effectiveness of the approach when being applied to real-world type errors.

5.2 Background on Python Type Checkers

In 2015, Python introduced a syntax for type annotations. These annotations are optional and not checked at runtime. The Python language also does not define a static type system, but leaves type checking to third-party tools. In response, the Python community has developed several type checkers that perform gradual type checking [231], i.e., a form of type checking aimed at exposing incompatibilities between the provided type annotations while allowing parts of the program to remain unannotated. Popular type checkers include Pyre, Mypy, Pytype, and Pyright.[46] The type systems implemented by checkers differ, and hence, different type checkers may reveal different type errors [211]. Conceptually, the approach described in this chapter is independent of a specific type checker and could be adapted to any of the popular checkers. Our implementation builds upon Pyre because it is widely used, available as open-source, backed by a major tech company, and has been the basis of recent work on studying Python type annotation practices [43]. Pyre reports a wide range of type-related problems, such as incompatible variable, parameter, and return types, uses of unbound names, unsupported operands, and inconsistent method overrides. We use Pyre's default configuration, i.e., it runs only on functions that are at least partially type-annotated. In the remainder of the chapter, we refer to Pyre using the term *type checker*.

[46]https://realpython.com/python-type-checking/

5.3 Preliminary Study

To guide important design decisions of our approach, we perform a preliminary empirical study that investigates three questions (PQs):

PQ1 *How repetitive are real-world type errors and type error fixes?* Answering this question is useful for deciding about the kind of technique, e.g., rule-based vs. data-driven, to build for automatically repairing type errors.

PQ2 *How difficult is identifying the fix location for a given type error?* Answering this question helps us decide how PyTy can effectively determine where in the given code to fix a type error.

PQ3 *How useful for fixing type errors are the error messages provided by a type checker?* Answering this question is useful to determine if and how a repair technique will be able to benefit from error messages.

5.3.1 Data Collection

To address the above questions, we systematically study type error fixes in the version histories of popular projects. We apply three strategies to select commits with type error fixes. First, we search for GitHub issues that call for help in fixing type errors. Second, we search for commits on GitHub via the keywords: "type+fix", "pyre" and "mypy" in Python repositories with more than 100 stars. Third, we use a dataset extracted from the top 10,000 Python repositories [43], which contains commits with edits related to inserting, removing, or updating a type annotation.

After collecting the commits, we clone the repositories and run the type checker before and after each commit. During our manual inspection, we observe that some warnings and fixes are not useful for our study towards building an APR tool, and hence, we remove (i) fixes that delete entire functions or files, without actually fixing a type error[47], (ii) import-related warnings, as they are often due to libraries missing in the type checker's search path, and (iii) fixes that add comments `# pyre-ignore` or `# type:ignore` to suppress warnings from the

[47]E.g., https://github.com/vkbottle/vkbottle/commit/2bc36b6

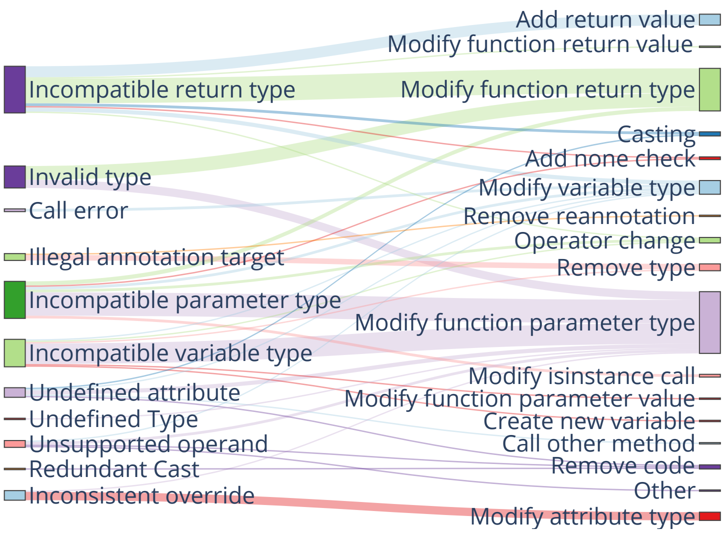


Figure 5.2: Type errors (left) and related fix patterns (right), based on 125 type error fixes collected in the preliminary study.

type checker. Overall, for the preliminary study, we collect 125 type error fixes from 14 GitHub repositories.

5.3.2 Results

PQ1: Repetitiveness of Type Errors and Fixes

We analyze the most frequent classes of type errors fixed by developers, which helps understand which errors concern developers the most, and hence, should be the focus of an APR technique. Figure 5.2 (left) shows the distribution of the most frequently fixed classes of type errors. We take the classes of type errors from the Pyre documentation.[48] The most frequent classes are incompatible return, variable, and parameter types, which together account for 64.8% of the dataset. For example, one such fix is for a function expected to return a `str` but

[48]https://pyre-check.org/docs/errors/

that actually returns `int` due to a statement `return -1`. The error is fixed by changing the return statement to `return "X"`.[49]

We also analyze the most frequent types involved in the fixes. We observe that Python's built-in types occur frequently, e.g., `str` (23.9%) and `int` (22.4%). Also relatively frequent are types related to optional values, such as `Optional` (9.3%) and `None` (5.6%), and other types from the `typing` library, e.g., `Union` (7.1%).

We study how type errors are fixed by manually categorizing the error-fixing code changes into 17 classes. This categorization was performed by one of the authors based on grounded theory [69], i.e., we discovered and refined fix patterns until they sufficiently covered the studied examples. Figure 5.2 (right) shows the distribution of the identified fix patterns. Beyond the distribution, the figure shows that there is no simple mapping from classes of type errors to fix patterns. The most frequent relationships are *Incompatible return type* fixed with the pattern *Modify function return type* (14.4%) and *Incompatible parameter type* fixed with the pattern *Modify function parameter type* (13.6 %). However, the same class of type error may also get addressed by applying several other fix patterns.

Summary: PQ1 A few kinds of type errors account for most fixed errors, and the fixes often involve Python's built-in data types. The fixes expose some recurring patterns, but only an ambiguous mapping from classes of type errors to fix patterns.

PQ2: Difficulty of Identifying the Fix Location

To assess the difficulty of localizing where to fix type errors, we start by investigating how much code developers typically change to fix a type error. Based on the categorization of fix patterns in Figure 5.2, we see that most fixes are single-line edits, such as modifying a type annotation from one type to another, changing an operator, removing a type annotation, or adding a cast. Next, we study the location of the fixes (Figure 5.4a). More than half of the fixes happen

[49] https://github.com/TheAlgorithms/Python/commit/97b6ca2

```
def is_valid_public_key_static(
  local_private_key_str: str,
        remote_public_key_str: str, prime:
        int
) -> bool:
  ...
[Error message] Expected ‘int‘ for 1st
      parameter but got ‘str‘.
  if pow(remote_public_key_str, (prime - 1)
         // 2, prime) == 1:
  ...
```

(a) Commit with a type error.

```
def is_valid_public_key_static(
  local_private_key_str: str,
  remote_public_key_str: int,
  prime: int
) -> bool:
  ...
[Fix pattern] Modify function parameter
       type.
  if pow(remote_public_key_str, (prime -
        1) // 2, prime) == 1:
  ...
```

(b) Commit that applies the “Use expected type” pattern.

```
def get_model_for_finetuning(
  previous_model_file: Optional[Union[Path,
         Text]]
[Error message] Expected 'Optional[Text]',
      got 'Union[None, Path, Text]'
) -> Optional[Text]:
```

(c) Commit with a type error.

```
def get_model_for_finetuning(
  previous_model_file: Optional[Union[
        Path, Text]]
[Fix pattern] Modify function return type
      .
) -> Optional[Union[Path, Text]]:
```

(d) Commit that applies the “Do not use expected type” pattern.

Figure 5.3: Examples of fixing type errors based on error messages.

exactly in the line where the type error is reported. Other locations include the function parameters, return annotations, and function callees (i.e., the functions that are called).

Summary: PQ2 Most fixes of type errors affect only a single line of code, which often is the line where the type checker reports the type error.

PQ3: Usefulness of Error Messages and Locations

Finally, we want to understand how useful the error messages provided by a type checker are for fixing type errors. To this end, we extract from the error message the kind of error, the types involved, and any hints about the location and the fix. We classify an error message as *correctly hinted* if the message contains the type that the developer uses to fix the error. Figure 5.3a shows an example, where the type checker returns the following error message:

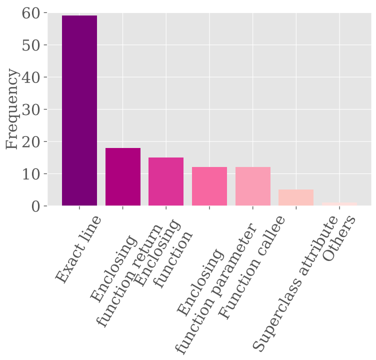
"Incompatible parameter type [6]: Expected `int` for 1st positional only parameter to call `pow` but got `str`", where the message hints at replacing `str` with the correct type `int`.[50] Note that the hinted type might not be an exact match to the newly annotated type. For example, we consider an error message that suggests `str` as correctly hinted also if the developer fixes the error using `Optional[str]`.

Given the above definition, we find that 89 out of 125 (71.2%) type fixes in our study are correctly hinted by the type checker. These hinted types can serve as a reference for APR tools to narrow down the search space. The types of code changes correctly hinted by the checker are shown in Figure 5.4b.

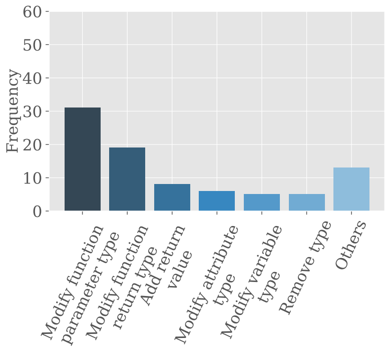
Figure 5.4c shows how the developers use (or do not use) the correctly hinted types. As an example, in Figure 5.3c, the type checker returns the following error message: "Incompatible return type [7]: Expected `Optional[Text]` but got `Union[None, Path, Text]`".[51] The developer does not fix the error by using the suggested type `Union[None, Path, Text]`, but instead uses `Optional[Union[Path, Text]]`. In contrast, the example in Figure 5.3b shows a case where the developer uses the type suggested by the type checker. We find that for 64 out of the 89 hints (71.9%) the type used by the developer is exactly as suggested in the error message. It is also common to introduce a value of the suggested type, e.g., by adding `return -1` to a function supposed to return `int`. Besides the 89 error messages that correctly hint at the correct type, most of the remaining messages (24 out of 36) give no hint at all. For example, this is the case for the error classes "Undefined type", "Invalid type", and "Undefined attribute".

Summary: PQ3 Most types used in fixes (71.2%) are correctly hinted by the type checker, and developers often follow these hints.

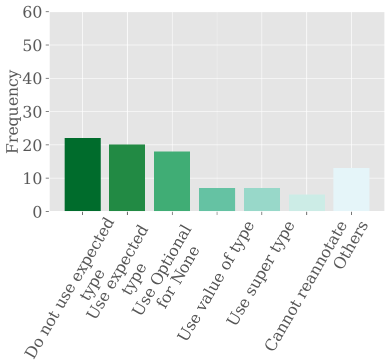
[50] https://github.com/TheAlgorithms/Python/commit/6089536
[51] https://github.com/RasaHQ/rasa/commit/1ded5ef



(a) Top seven locations for fix patterns.



(b) Top seven fix patterns correctly hinted by the type checker.



(c) How developers use the hint from the type checker.

Figure 5.4: Fix locations and usefulness of error messages.

5.3.3 Implications

The three main findings of the preliminary study guide the design of our approach as follows.

- PQ1 We observe that type errors and their fixes expose recurring patterns, which might suggest an approach based on manually designed rules and heuristics for selecting them. However, we also find that there are ambiguous mappings from errors to fix patterns, making a rule-based approach laborious and fragile. As a result, we decide against a rule-based and in favor of a data-driven approach, aiming for a model that learns when to apply which fix pattern from fixes performed by developers.
- PQ2 We find that most type errors are fixed by editing a single line, and that this line is often localized correctly by the type checker. Hence, we focus our work on fixing type errors in single-hunk edits[52] and exploit the localization hint given by the type error location.
- PQ3 We find that the error message provided by the type checker often gives valuable hints for finding the fix, e.g., which type to use. As a result, we provide the error message as an input to our approach.

5.4 Approach

Based on the findings of our preliminary study, we design PyTy, a data-driven approach to automatically fix static type errors in Python using a cross-language transfer learning approach. Figure 5.5 shows an overview of the approach, which consists of two phases. First, during the offline phase, we automatically collect a dataset of type error fixes, PyTyDefects, from GitHub by combining delta debugging and gradual type checking, followed by fine-tuning a pre-trained model [19] with PyTyDefects. Second, during the online phase, PyTy receives code with a type error as the input and then queries the model for fix candidates. The approach uses the type checker to validate that the type error gets resolved when applying a fix candidate, and then reports only fixes that are guaranteed to remove the targeted type error.

[52]Hunks may be larger than single lines, allowing PyTy to predict some fixes that involve multiple lines.

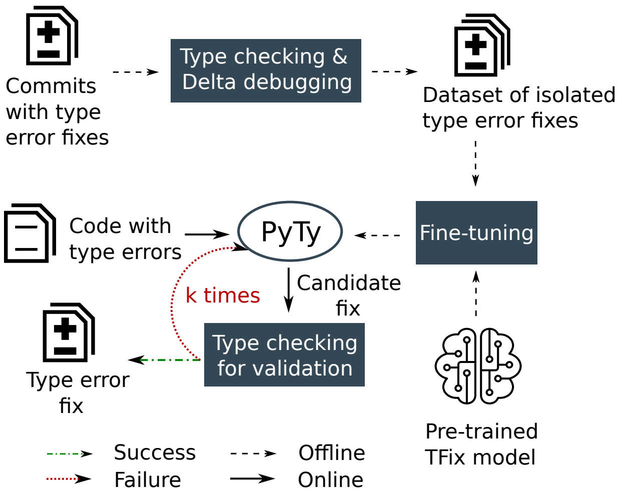

Commits with type error fixes
Type checking & Delta debugging
Dataset of isolated type error fixes
Code with type errors
PyTy
Fine-tuning
k times
Candidate fix
Type checking for validation
Type error fix
Pre-trained TFix model
Success
Failure
Offline
Online

Figure 5.5: Overview of the approach.

5.4.1 Automated Data Gathering

To build a learning-based APR model, we must first collect a relevant dataset as our training data. As a first step, we search for Python repositories that are popular (≥ 100 stars), have a manageable size (≤ 5GB), and were created between 2010 and 2021 on GitHub. We use the keywords "fixing+typing", "fixing+pyre", "fixing+mypy", "typing+bug", and "typing+error" to search for commits that possibly remove type errors. Next, we run the type checker before and after each such commit to find commits that indeed remove type errors. As a result, we obtain 32,330 type errors that are removed by 4,515 commits in 176 GitHub repositories.

Many of the extracted commits contain changes not directly related to fixing type errors. Moreover, a single commit often fixes multiple type errors. Figure 5.6 illustrates these problems with an example.[53] To isolate individual type error fixes, we present a delta debugging-inspired [284] algorithm that reduces commits into small code changes that fix a single type error. The basic idea is to

[53] Simplified from https://github.com/jazzband/django-redis/commit/5f6f383

iteratively reduce the set of code hunks while preserving the fact that the code change fixes a particular type error.

Algorithm 5.1 summarizes our approach for reducing a commit to a small set of code changes that fix the given type error. We illustrate the algorithm using the example in Figure 5.6. We focus on the type error "Unbound name: `basestring` is used but not defined in the current scope", reported for the line in hunk H1. The error gets fixed by changing the base class to `object`. Our approach considers the four code hunks (H1, H2, H3, and H4) of this commit, and determines which hunks are relevant for fixing the type error with the following steps, where line numbers refer to Algorithm 5.1:

1. The algorithm splits the set of hunks into `H1+H2` and `H3+H4` with granularity two (line 8):
    a) The algorithm patches the code with only hunks `H1+H2`, which yields parsable code (lines 9 and 10).
    b) The type error disappears and there are no new errors (lines 11 and 12).
    c) There are still two hunks `H1+H2` (line 13).
2. The algorithm splits the code hunks `H1+H2` into `H1` and `H2` (line 8):
    a) The algorithm patches the code with only hunk `H1`, which yields parsable code (lines 9 and 10).
    b) The type error disappears and there are no new errors (lines 11 and 12).
3. The algorithm returns `H1` as a minimal code change to fix the type error (line 14).

To properly track the error location while reducing the hunks, we need to keep track of how the line numbers change. To this end, we calculate the new line number based on how many lines are inserted or removed in each code hunk. We consider the error fixed if the error no longer exists at the corresponding line and column. If the error is located inside a code hunk, i.e., the code with the error is being modified, we consider the error as fixed only if all lines in the code hunk are free of errors after the change.

```
# Hunk H1
class CacheKey(basestring):

# Hunk H2
  pass



# Hunk H3
  if isinstance(key, CacheKey):
    key = CacheKey(smart_str(key))

# Hunk H4
  if timeout == 0:
```

(a) Commit 1.

```
# Hunk H1
class CacheKey(object):

# Hunk H2
  def __init__(self, key):
    self._key = key
    ...

# Hunk H3
  if not isinstance(key, CacheKey):
    key = CacheKey(key)

# Hunk H4
  if timeout is None:
    ...
```

(b) Commit 2.

Figure 5.6: Multi-hunk commit that fixes multiple type errors.

To ensure the quality of PyTyDefects, we apply additional filtering steps. Algorithm 5.1 checks that there are no new errors introduced by the code changes (line 12). The algorithm also rejects any set of code hunks that result in parsing failures (line 10). The space complexity of the algorithm is $\mathcal{O}(2 * N)$, where $N = size(D_{original})$ and the time complexity is $\mathcal{O}(N * \log N)$.

Running the algorithm on the 32,330 type errors gives 11,955 examples of reduced error fixes. We further filter them by keeping only fixes that (i) are relatively small ($\leq$ 512 characters and at most three changed lines), which is motivated by limitations of the neural model (Section 5.4.2); (ii) do not contain any error suppression; (iii) are not only deletion; (iv) are located close (i.e., within the same hunk) to the reported bug location. Finally, after applying these filters, PyTyDefects has 2,766 entries that cover ten frequent categories of type errors listed in Table 5.1.[54] We select 10% (always rounding up to the nearest integer) of the entries of each error class in this final dataset as a test set, and then split the remaining fixes into 90% for training and 10% for validation.

[54]The distribution of type errors is similar to that in our preliminary study (Figure 5.2), but not exactly the same because the datasets differ.

5.4.2 Neural Type Error Fixing

Given the automatically extracted dataset of type error fixes, PyTy trains a neural model that predicts how to fix type errors. The input to the model is a sequence of one or more lines, i.e., the size of a single hunk, that contains a type error. The output of the approach is a fix that removes the targeted type error.

Base Model. Instead of learning a model from scratch, we fine-tune a model pre-trained on another APR task. Building on a pre-trained model is motivated by the fact that PyTyDefects, with 2,766 examples, is relatively small. As our base model, we use TFix [19], a learning-based APR technique trained to fix linter errors in JavaScript. We select TFix for three reasons: (i) it is already trained on a bug fix dataset of 104,804 samples, (ii) it accepts error messages as input, and (iii) the TFix authors used it to predict single line fixes, which resembles our single-hunk setup. By fine-tuning TFix, PyTy transfers the already learned knowledge to the related but different domain of Python type errors (Section 5.6.3). TFix itself is based on T5 [207], a transformer-based model that maps sequences of input tokens to sequences of output tokens. The simple input and output structure eliminates the need for implementing a static analysis tool to transform our code into a specific structure, such as graphs [5, 45]. Furthermore, having an unconstrained token sequence may enable the model to fix errors missed by a template-based APR approach, which is inherently limited to its set of templates (Section 5.6.4).

Fine-Tuning for Python Type Error Fixing. To fine-tune TFix with PyTyDefects, we follow the same input format as TFix:

$$\text{“fix”} \sqcup t \sqcup m \sqcup l_k \sqcup \text{“:”} \sqcup C$$

where “fix” and “:” are literals, t is the class of type error, m is the error message, l_k is the line of code with the type error, $\sqcup$ represents a space, and C represents the buggy lines of code (i.e., the single-hunk we extracted in Section 5.4.1). In the T5 framework, the string "“fix” $\sqcup t \sqcup m \sqcup l_k \sqcup$ “:”" represents the current task, and C represents the input of this task. The model outputs C', which we use as a replacement for C to fix the type error.

Python Code Pre- and Post-Processing. We use the tokenizer from the Python standard library to pre-process the source code and inject special tokens for indentation and dedentation. TFix uses SentencePiece [127] as its tokenizer. However, SentencePiece does not take the number of whitespaces into account, as it escapes all whitespaces into a single "_" symbol. Since the amount of whitespace carries semantics in Python, we preserve this information by adding special tokens "<IND>" and "<DED>" into the source code before passing it to the model. Given a prediction by the model, PyTy removes the special tokens in a post-processing step to obtain syntactically correct Python code.

Validating Fixes via Type Checking. Once trained, we query the model for a ranked list of the k most likely fixes. To ensure that a fix suggestion given to a user indeed removes the targeted type error, PyTy validates all candidate fixes by running the type checker on them. If and only if the targeted type error disappears and no new errors appear, the fix is suggested to the user.

Algorithmus 5.1 Extract relevant hunks with delta debugging.

```
Input: Files $f_{old}$ with type error $err$ and $f_{new}$ after commit with $err$ fixed
Output: Minimal hunk(s) of the commit, containing only $err$ fixed
 1: $W \leftarrow type\_check(f_{old})$                                  ▷ Set of all warnings in $f_{old}$
 2: $D_{original} \leftarrow diff(f_{old}, f_{new})$                     ▷ All diff hunks between $f_{old}$ and $f_{new}$
 3: $granularity \leftarrow 2$                                          ▷ Set default granularity
 4: while $granularity \leq size(D_{original})$ do
 5:     $min \leftarrow False$
 6:     $D \leftarrow D_{original}$
 7:     while $size(D) > 1$ and $granularity > 1$ do
 8:         for $d$ in $split(D, granularity)$ do                        ▷ Split the set of hunks $D$
 9:             $f_{fixed} \leftarrow patch(f_{old}, d)$                 ▷ Apply a subset of $D$ to file $f_{old}$
10:             if $parsable(f_{fixed})$ then
11:                 $W_{fixed} \leftarrow type\_check(f_{fixed})$
12:                 if $\nexists$ $err$ in $W_{fixed}$ and $W_{fixed} == W$ then
13:                     if $size(d) == 1$ then
14:                         return $d$
15:                     else
16:                         $D \leftarrow d$
17:                         $min \leftarrow True$
18:                         break
19:                     end if
20:                 end if
21:             end if
22:         end for
23:         if $min == False$ then
24:             if $granularity * 2 \leq size(D)$ then
25:                 $granularity \leftarrow granularity * 2$
26:             else if $granularity == size(D)$ then
27:                 return D
28:             else
29:                 $granularity = size(D)$
30:             end if
31:         end if
32:     end while
33:     $f_{fixed} \leftarrow patch(f_{old}, D)$                         ▷ Apply $D$ (size=1) to file $f_{old}$
34:     if $parsable(f_{fixed})$ then
35:         $W_{fixed} \leftarrow type\_check(f_{fixed})$
36:         if $\nexists$ $e$ in $W_{fixed}$ and $W_{fixed} - W = \emptyset$ then
37:             return $D$
38:         end if
39:     end if
40: end while
```

5.5 Implementation

We fine-tune the t5-base (220M parameters) model of TFix for 30 epochs with a batch size of 32, and then evaluate the model with the best validation loss on the validation set. The model converges at the 17th epoch. We follow the default hyperparameters of TFix [19].

When validating candidate fixes using the type checker, we sample from the model up to $k = 50$ predictions to be validated. Since PyTy validates fix candidates automatically, a user does not have to inspect these 50 suggestions, but only the first one found to successfully remove the type error. To fix 281 type errors (i.e., our test set, which amounts to a total of 174,586 lines of code) and automatically check whether the targeted type errors disappear, PyTy takes in total six hours and 44 minutes, i.e., an average of 86.2 seconds per type error fix. We perform all experiments on a server with 48 Intel Xeon CPU cores clocked at 2.2GHz, 250GB of RAM, one NVIDIA Tesla V100 GPU, running Ubuntu 18.04. Most of the time is spent on running the type checker for validating candidate fixes.

5.6 Evaluation

We evaluate both PyTyDefects and the learning-based type error repair, focusing on the following research questions (RQs):

RQ1 How effective is our automated data gathering at producing minimal code changes that fix type errors?

RQ2 How effective is PyTy at fixing type errors?

RQ3 How do variants of PyTy compare to the full approach?

RQ4 How does PyTy compare to state-of-the-art APR techniques?

5.6.1 RQ1: Effectiveness of Automatic Data Gathering

Data analysis. To validate the effectiveness of automatically gathering PyTy-Defects, two of the authors independently annotate a random sample of 100 of the 2,766 entries in the dataset. Each entry is assigned one of three labels:

```
basis_from: Basis = None,
basis_to: Basis = None,
I: ndarray = None,
expand: bool = False) -> ndarray:
```

(a) Commit with multiple type errors.

```
basis_from: Optional[Basis] = None,
basis_to: Optional[Basis] = None,
I: Optional[ndarray] = None,
expand: bool = False) -> ndarray:
```

(b) Commit with multiple type error fixes.

Figure 5.7: Example of a correct (but not minimal) entry in PyTyDefects.

minimal if the extracted code change fixes a type error and cannot be further reduced, *correct but not minimal* if the extracted code change correctly fixes a type error but is not minimal, and *wrong* otherwise.

Results. After independently labeling the 100 entries, the two annotators initially have an intersection of 89 labels. After discussing the divergent labels, the labels of two entries are refined, giving a final agreement on 94/100 *minimal*, 3/100 *correct but not minimal*, and 0/100 *wrong* entries. The remaining three entries with divergent labels are due to hunks that fix two type errors at once. These entries are minimal in the sense that a hunk-based reduction algorithm cannot further reduce them, but they could be further reduced by a more fine-grained reduction algorithm [93, 245]. The inter-rater agreement, as given by Cohen's kappa coefficient [38] is 0.651, which means a *substantial agreement* [128].

As an example of a *minimal* type error fix, recall hunk `H1` from the previously discussed commit in Figure 5.6. All changes in hunk H1 are necessary for fixing the type error. Figure 5.7 shows an example of a *correct but not minimal* reduced commit, which includes some changes not relevant to fixing the type error.[55] A single hunk updates multiple parameter type annotations of the same function. However, only one code change is relevant to fixing the type error reported for `basis_to`, which should be annotated `Optional[Basis]` instead of `Basis` as it is initialized to `None`.

[55] https://github.com/kinnala/scikit-fem/commit/a555ca3

```
...
vprint(f"{prefix} {lineno}: {
     action}
  Constrain Mouse: {'y' if
        constraint > 0
  else ('no' if constrained
        == 0 else 'check
        stack')}")
```

(a) Code with type error.

```
...
vprint(f"{prefix} {lineno}:
       {action}
  Constrain Mouse: {'y' if
        constraint > 0
  else ('no' if constraint
        == 0 else 'check
        stack')}")
```

(b) Fix by the developer.

```
...
vprint(f"{prefix} {lineno}:
       {action}
  Constrain Mouse: {'y' if
        constraint > 0
  else ('no' if constraint
        == 0 else 'check
        stack')}")
```

(c) Fix suggested by PyTy.

Figure 5.8: Exact match of fix for type error "Unbound name: Name `constrained` is used but not defined in the current scope".

```
...
string = _fmt(string)
return lib.TCOD_console(
  self.console_c, x, y,
        width, height,
  string
)
```

(a) Code with type error.

```
...

return lib.TCOD_console(
  self.console_c, x, y,
        width, height,
  _fmt(string)
)
```

(b) Fix by the developer.

```
...
byte_string = _fmt(string)
return lib.TCOD_console(
  self.console_c, x, y, width
        , height,
  byte_string
)
```

(c) Fix suggested by PyTy.

Figure 5.9: Correct fix different from the developer-provided fix for type error "Incompatible variable type: `string` is declared to have type `str` but is used as type `bytes`".

```
},
F5_DEVICE_TYPE: {
  DEVICE_CLASS_KEY:
        F5Device,
```

(a) Code with type error.

```
},
F5_API_DEVICE_TYPE: {
  DEVICE_CLASS_KEY:
        F5Device,
```

(b) Fix by the developer.

```
},
DEVICE_TYPE: {
  DEVICE_CLASS_KEY:
        F5Device,
```

(c) Fix suggested by PyTy.

Figure 5.10: Fix predicted by the neural model, but not suggested to the user, as the type error "Unbound name: Name `F5_DEVICE_TYPE` is used but not defined in the current scope" would still exist for `DEVICE_TYPE`.

```
global Bot
if self is Bot:

Bot = new
```

(a) Code with type error.

```
global Bot
if self is Bot:
assert isinstance(new,
       BotUser)
Bot = new
```

(b) Fix by the developer.

```
global Bot
if self is Bot:

new_Bot = new
```

(c) Fix suggested by PyTy.

Figure 5.11: PyTy-suggested fix that removes the error “Incompatible variable type: `Bot` is declared to have type `BotUser` but is used as type `User`”, while changing the behavior in an unintended way.

Summary: RQ1. The automated data gathering yields type error fixes that are mostly correct (97/100) and minimal (94/100), i.e., PyTyDefects provides a solid basis to train and validate PyTy.

5.6.2 RQ2: Effectiveness of PyTy

We evaluate the effectiveness of PyTy on all ten classes of type errors covered by our test set. We configure the approach to consider up to $k = 50$ candidate fixes. Note that users do not have to manually check all candidate fixes, but only see the first successful fix.

Metrics. We use two metrics to evaluate the effectiveness of PyTy. First, we compute the *error removal rate*, i.e., how often the approach succeeds at finding a fix that removes the targeted type error without introducing new type errors. Second, we compute the *exact match rate*, i.e., how often the model output is identical to the fix committed by the developer. This metrics underapproximates the abilities of PyTy, as there might be fixes that address the type error in a reasonable way that differs from the original fix.

Quantitative Results. Table 5.1 shows the number of samples used for training and testing, the error removal rate, and the exact match accuracy. Each row in the table corresponds to one kind of type error reported by the type checker. PyTy successfully removes the type error in 85.4% of the cases, and it finds

Table 5.1: Results of PyTy for each class of type error.

Classes of type errors	Samples	(test set)	Effectiveness of PyTy: Error removal	Effectiveness of PyTy: Exact match
Incompatible variable type	821	(83)	90.4%	65.1%
Incompatible parameter type	600	(60)	80.0%	36.7%
Incompatible return type	296	(30)	73.3%	43.3%
Invalid type	291	(30)	100.0%	83.3%
Unbound name	258	(26)	76.9%	42.3%
Incompatible attribute type	258	(26)	92.3%	73.1%
Unsupported operand	124	(13)	76.9%	38.5%
Strengthened precondition	59	(6)	83.3%	50.0%
Weakened postcondition	51	(6)	50.0%	0.0%
Call error	8	(1)	100.0%	100.0%
Total	**2,766**	**(281)**	**85.4%**	**54.4%**

exactly the developer-provided fix for 54.4% of all errors. Comparing different kinds of type errors, we find the approach to be effective across a wide range of errors. An exception are *Weakened postcondition* errors, which are often caused by type-incorrect, overriding methods in custom classes, i.e., a kind of mistake that requires non-local, project-specific information to be fixed.

Examples. The example in Figure 5.9 fixes the type error in a way that matches the intention of the developer but differs from the original fix.[56] The developer fix directly passes the byte string `_fmt(string)` as an argument to the function `lib.TCOD_console_printf_ex`, avoiding the error caused by re-assigning the byte string to the variable `string`, which is previously annotated as type `str`. The PyTy-suggested fix instead declares a new variable `byte_string` for the byte string, and passes it to `lib.TCOD_console_printf_ex` as an argument.

Figure 5.10 shows a predicted fix that fails to remove the type error.[57] The developer fix uses a variable (`F5_API_DEVICE_TYPE`) imported from another

[56] https://github.com/libtcod/python-tcod/commit/60066f3
[57] https://github.com/networktocode/pyntc/commit/ebb35344e0121

package. However, since the context code and the error message do not give any hint about the identifier to use, the model simply replaces it with `DEVICE_TYPE`. Because PyTy validates that a fix candidate removes the type error before reporting the fix to the user, this fix suggestion is not shown to users, highlighting the importance of validating fix candidates.

Finally, Figure 5.11 fixes the type error but changes the semantics of the code in an unintended way.[58] The error is because `Bot` and `new`, which is a variable, have incompatible types. The developer fixes the error by asserting that `new` is of type `BotUser`. PyTy instead suggests a fix that declares a new variable `new_Bot`, which however fails to update the global `Bot` variable. We include this example to show that PyTy is limited by relying on the type checker as the only validation mechanism. Future work could address this limitation by additionally validating fixes by running a test suite.

Type Fixes in the Wild. To further validate the usefulness of PyTy in practice, we create pull requests with PyTy-suggested fixes for type errors. We run Pyre on different GitHub projects randomly picked among the projects in PyTyDefects. In total, we create 30 pull requests (for 17 incompatible variable type errors, ten incompatible parameter type errors, and three invalid type errors). By the time of this writing, 20 of the pull requests have been merged, six are still open, and four are closed.

For the pull requests merged so far, the developers generally were grateful about the changes. In one case, the developers even asked us to apply similar fixes in other code locations, which we did, as we could use PyTy-suggested fixes there as well. The four closed pull requests are: (i) Two cases where the developers prefer to use type casts and dynamic type checks rather than updating the type annotations; (ii) One case where the developers decided to suppress a warning about an incompatible `Optional` variable type; and (iii) One case where the developers consider a warning about an incompatibility between `List[Optional[Path]]` and `List[None]` to be a false positive. Overall, the developers' feedback confirms PyTy's usefulness in practice.

[58] https://www.github.com/lykoss/lykos/commit/abbd35c

Table 5.2: Ablation study and comparison with LLMs.

	Error removal (%)			Exact match (%)		
Approach	Top-1	Top-5	Top-50	Top-1	Top-5	Top-50
No pre-training	47.3	57.3	71.2	30.2	45.2	48.8
Vanilla TFix	4.6	11.0	16.7	0.0	1.1	1.8
No preprocessing	17.8	23.5	29.5	37.0	45.6	54.1
Small TFix model	43.1	63.3	79.0	32.7	44.8	53.0
text-davinci-003	21.7	27.8	34.6	14.6	18.1	20.9
gpt-3.5-turbo	21.9	23.8	26.0	10.3	12.1	14.5
gpt-4	34.1	36.7	39.1	18.9	22.1	26.4
Full PyTy	**50.9**	**66.2**	**85.4**	**37.7**	**48.0**	**54.4**

Summary: RQ2. PyTy successfully removes the type error in 85.4% of the cases evaluated, and it finds exactly the developer-provided fix for 54.4% of all errors.

5.6.3 RQ3: Ablation Study of PyTy

We perform an ablation study to evaluate the effectiveness of PyTy in different configurations. Table 5.2 summarizes the results discussed in the following.

No pre-training. We train the T5 model directly on PyTyDefects, i.e., without pre-training the model on the JavaScript APR tasks. The purpose of this experiment is to check if the knowledge of fixing JavaScript errors helps in fixing Python type errors. We use the same experimental setup as discussed in Section 5.5, except that training continues beyond 30 epochs because the evaluation loss keeps decreasing. We train the model for 100 epochs and pick the model with the least validation loss, which is at the 32nd epoch. The results show that pre-training the model on the JavaScript repair task contributes significantly to its effectiveness. For example, the top-1 exact match rate drops from 37.7% to 30.2% without pre-training.

Vanilla TFix. We try to predict the fix with the original TFix model, i.e., without fine-tuning TFix with PyTyDefects. The purpose of this experiment is to check whether gathering a dataset of type errors is really necessary. We use the same experimental setup as discussed in Section 5.5, except that we use the t5-large (770M parameters) model of TFix. The reason is that removing fine-tuning also removes the resource constraints that motivated us to use the t5-base model (220M parameters). For this experiment, we do not preprocess the Python source code as the tokenizer of the TFix model is trained without the special tokens. As shown in Table 5.2, the effectiveness drops dramatically, e.g., to only 1.8% exact matches within the top-50 suggestions. The reasons are (i) that Python and JavaScript have different syntax, i.e., it is unlikely for the model to output syntactically correct Python code, and (ii) that the TFix model is not trained to fix type errors.

No preprocessing. We try to generate the fix without the preprocessing that adds indentation and dedentation tokens (Section 5.4.2). We use the same experimental setup as discussed in Section 5.5, but we remove the special tokens from the input and output code. We find preprocessing to be important, as otherwise the error removal rate drops significantly, e.g., from 50.9% to 17.8% in the top-1 prediction. For exact match accuracy, the decrease in effectiveness is less strong, but the exact match might not be equal to the actual developer fix, as we ignore the newline tokens and the number of whitespaces for the comparison.

Small TFix model. To study the impact of the model size, we try to predict the fix by basing PyTy on the small TFix model (with only 60M parameters). We use the same experimental setup as discussed in Section 5.5. As the evaluation loss of this model keeps decreasing beyond the 30th epoch, we train the model for 100 epochs, which converges at the 47th epoch. The effectiveness of PyTy is negatively affected by using a smaller model, e.g., with 43.1% instead of 50.9% top-1 error removal rate. At the same time, the negative impact of the small model can be partially compensated by considering more fix suggestions: For example, the top-50 exact match rate is reduced only slightly from 54.4%

to 53.0%. These results show that PyTy could also be effective in a resource-constrained setup, such as a developer laptop instead of a server.

Summary: RQ3. The full PyTy outperforms simpler variants of the approach, showing that each of PyTy's components contributes to its effectiveness.

5.6.4 RQ4: Comparison with Prior Work

RQ4a: PyTy vs. Large Language Models

Fixing type errors relates to general-purpose APR [134]. The following compares PyTy with large language models (LLMs), which have been shown to yield state of the art results [108, 272, 273]. PyTy and LLMs fundamentally differ in the sense that PyTy is designed and fine-tuned specifically for type error repair, whereas LLMs are trained in a task-independent manner, but typically on much more data.

Experimental Setup. We compare PyTy with three recent models offered by OpenAI: *text-davinci-003*, *gpt-3.5-turbo*, and *gpt-4*. Our prompt consists of five parts: a description of the task, the buggy code snippet, the type checker's error message, the line containing the error, and a description of the expected output format.

Results. The lower part of Table 5.2 shows the effectiveness of different models. PyTy clearly outperforms all LLMs in terms of error removal and finding the exact developer fix. The gpt-4 model, as the most recent and largest model, is the most effective LLM. The text-davinci-003 model is slightly more effective than gpt-3.5-turbo, which may be because the latter is optimized for chat. Manually analyzing the successful fixes, we notice that the LLMs mostly fix those errors that can be fixed with a single-token edit. Instead, PyTy can fix more complex type errors.

Summary: RQ4a. PyTy is more effective than prompting general-purpose LLMs (54.4% vs. 26.4%).

RQ4b: PyTy vs. vs. PyTER

PyTER [186] repairs bugs that manifest through a `TypeError` exception. For a comparison, consider the two subproblems that both approaches address. Subproblem 1 is *detecting a type error*, done by the static type checker in our approach and by observing a runtime exception in PyTER. Subproblem 2 is *fixing a detected type error*, done by a neural model in our approach and by applying a set of repair templates in PyTER. How PyTy and PyTER address subproblem 1 differs fundamentally. While static type errors manifest without running the code, revealing a runtime type error require tests cases or a production run that triggers the error. Moreover, the same conceptual problem may manifest at different locations. For example, a function that returns an incorrect value will manifest as a static type error at the return statement, but as a runtime type error at a code location that uses the value. Because of these differences, performing a direct, end-to-end comparison is neither possible nor meaningful. Instead, we quantify the overlap of the two approaches in terms of the errors they address and the fixes that they could potentially find, which answers four questions.

PyTER on PyTyDefects. *1) How many of the errors in PyTyDefects manifest via a runtime type error?* We pick a random sample of 30 of all 281 fixes in our test set and inspect their commit messages. The inspection shows that for 16/30 fixes, the problem was certainly found via static type checking, e.g., because the message mentions the type checker, and for 27/30 fixes, the problem was certainly not found via a `TypeError` thrown at runtime. *2) How many of the type errors in PyTyDefects are in the scope of PyTER's fix templates?* The repair templates cover three kinds of fixes: adding an `instanceof` check, adding a type conversion, e.g., via a call to `int()`, and adding code to catch and handle a `TypeError` exception. We check for each type error in our test set whether PyTER's repair templates can be instantiated into the fix, which shows that 15/281 type errors are in scope for PyTER, whereas the remaining 266 errors are not covered by any repair template. Examples of fixes that are out-of-scope for PyTER are: (i) Fixes that change a value, e.g., by modifying a string `"a b c"` into an array of strings `["a", "b", "c"]`. (ii) Fixes that change a type annotation, e.g., from `T` to `Optional[T]`. (iii) Fixes that add a call to `typing.cast()`. In summary, PyTER address only a small fraction of the type errors in our dataset.

PyTy on PyTER's dataset. *3) How many of the errors in PyTER's dataset manifest via a static type error?* The Pyre type checker that PyTy builds on checks (partially) type-annotated code only. Among the 93 errors in PyTER's dataset, 16 are in a type-annotated function, and hence, checked at all, but the type checker does not find the errors fixed by PyTER. *4) How many of the type errors in PyTER's dataset are in the scope of PyTy's neural model?* Our approach focuses on single-hunk fixes where the type error location is inside the hunk that needs to be changed. While these assumptions commonly hold for static type errors (Section 5.3), only 11/93 errors in PyTER's dataset match the assumptions. Similar to above, PyTy address only a small fraction of the type errors in the PyTER dataset.

Summary: RQ4b. Our approach and PyTER [186] are complementary in the sense that they address type errors that manifest in different ways and that they apply different kinds of fixes.

5.7 Discussion and Threats to Validity

5.7.1 Python Repositories

We select popular projects for our dataset, because recent work finds such projects to contain type annotations and type errors [43]. A different set of repositories could yield different results, in particular for the preliminary study (Section 5.3).

5.7.2 Limitations of static type checking

PyTy builds upon the Pyre type checker, which as all static type checkers, may suffer from false positives and false negatives. A false positive, where the type checker incorrectly reports a type error in correct code, may lead to unnecessary code modifications by PyTy. Conversely, a false negative, where an error goes unnoticed by the type checker, may cause PyTy to suggest a fix that does not really solve the problem, or even worse, introduce a new problem. As a lower bound on PyTy's effectiveness despite these limitations, we find that 54.4% of the predicted fixes exactly match the developer's fix. Other type checkers than Pyre may find different kinds of type errors and provide different kinds of hints for fixing them. Because our approach uses the type checker as a black-box, adapting our implementation to support another type checker seems straightforward.

5.7.3 Type annotations

Because the type checker reports errors only in functions that are at least partially type-annotated, PyTy cannot fix errors in completely unannotated code. Despite this limitation, there is evidence that more and more code bases get type-annotated, and hence, are in scope for PyTy. For example, a recent study on the evolution of type annotations [43] finds 50 type annotations per 1,000 lines of code and an increasing trend on the adoption of type annotations. Moreover, our dataset of thousands of real-world commits that address type errors shows that developers care about such errors. Finally, as described in Section 1, large companies, such as Google, Dropbox, and Meta, are actively working toward type-annotating their Python code bases.

5.7.4 Type Errors

PyTyDefects, containing 2,766 real-world type error fixes, is filtered to contain only errors fixable with a single-hunk code change, and we cannot draw any conclusions about more complex fixes. As shown in Section 5.3.2, many real-world fixes are local edits, which has motivated our design decision to focus on single-hunk fixes. The distribution of error classes in PyTyDefects reflects the errors that occur in practice, and does not cover all error classes that the type checker may find. Thanks to the data-driven design of PyTy, the approach should be able to fix further classes of type errors when given corresponding training data.

Future work We plan to improve the error localization and will try different prompts to improve the performance of LLMs. Moreover, we plan to fine-tune different models beside TFix and apply PyTy to more classes of type errors. Finally, we plan to integrate our approach into an IDE.

5.8 Concluding Remarks

This chapter presents PyTy, the first automated repair technique targeted specifically at statically detectable type errors in Python. The design of the approach is motivated by the findings of a preliminary study. To generate a relevant dataset, we apply a combination of delta debugging and type checking, which results in PyTyDefects, containing 2,766 Python type errors and fixes. We then present cross-lingual transfer learning, which addresses the problem of having a small dataset for a deep learning model by fine-tuning an existing APR model originally trained for another task and language. Our evaluation shows the effectiveness of PyTy, e.g., by providing a fix that removes the targeted type error for 85.4% of the studied errors. Finally, as of this writing, 20 out of 30 GitHub pull requests based on PyTy type fixes have been merged by developers, demonstrating the usefulness of PyTy in the wild.

Chapter 6
Related work

In this chapter, we explore research closely related to the four projects described in this dissertation. The related work presented here is not meant to be exhaustive; instead, it underlines significant works on related problems and compares this dissertation to existing work.

6.1 Analyses of Code Changes

There are numerous approaches that try to analyze and understand code changes. Hashimoto et al. propose a technique for reducing a diff to the essence of a bug [90]. Nielsen et al. [185] use JavaScript code change templates to fix code broken due to library evolution. Additionally, some methods document code changes with natural language descriptions [26]. Predictive models like SCC [67] and DeepJIT [97] estimate the likelihood of a code change introducing bugs. Approaches like DiffBase [270] aid in multi-version program analyses, while CodeShovel[79] chronicles a method's evolution within version histories. All these approaches relate to our work in Chapter 3 by also reasoning about code changes, but they aim for different goals than DiffSearch, which is a search engine for code changes.

Moreover, several researchers focus on abstract representations, constructing edit scripts on ASTs [56, 58, 61, 89], providing an abstract representation of a change that can then be applied in different scenarios [167]. Future work for DiffSearch could explore using an edit script-based representation of code changes to search for code changes. An advantage of our parse tree-based feature extraction in Chapter 3 is that it does not require aligning the old and new code, allowing us to featurize hundreds of thousands of code changes in reasonable time. Paletov et al. [190] study code changes related to crypto APIs and they extract security fixes from code histories. Weissgerber et al. [264] identify code changes that have a high chance to be refactored. Hashimoto et al. propose a technique for reducing a diff to the essence of a bug [90].

SCC [67] and DeepJIT [97] are predictive models that estimate the correlation between the insertion of a code change and introducing a bug. A related problem is to find the bug-inducing code change for a given bug report [265, 269]. Finally, Zeller [284] introduces the delta debugging algorithm to find code hunks that are "failure-inducing" in a commit. The algorithm is widely used, in particular for fault localization [267]. In Chapter 5, we treat code hunks that fix the type errors as "failure-inducing", in an approach similar to prior work [17], but adopted to type errors.

6.2 Software Evolution Studies

Software is continuously evolving and many researchers perform interesting studies. Nguyen et al. [181] study the repetitiveness of code changes in code histories, modeling a code change as a pair of AST sub-trees within a method. Gu et al. [81] analyze large project histories to study problems related to multi-thread programming. Dagenais et al. [39] study code evolution to recommend relevant changes with a high precision. Chapter 4 contributes the first in-depth study of the evolution of type annotations in Python.

6.3 Tracking Code Elements Across Version Histories

Tracking code elements across the different commits in a project is a challenging problem due to the various ways how code may change, and because there is no universally accepted definition of when a code element remains “the same” across a change. Grund et al. [79] propose CodeShovel, which addresses this problem on the method level through an AST-based, heuristic algorithm. Ketkar et al. [119] focus on type changes in Java, using type fact graphs to represent code changes. In contrast, our algorithm in Chapter 4 for extracting type annotation changes is the first attempt at tracking annotations in a dynamically typed language (Python).

6.4 Mining and Learning from Code Changes

Mining code repositories has unveiled development histories as potent knowledge wells. Approaches vary from extracting repetitive code changes [176, 178, 179], predict code changes [253], predict bugs [125, 150], or to learn about API usages [177, 190] as mentioned in the previous Section. Mining approaches typically consider all code changes in a project’s version history or filter changes using simple patterns, e.g., keywords in commit messages. In contrast, DiffSearch allows for identifying code changes that match a specific query.

Large sets of code changes enable learning-based techniques. One line of work learns from specific kinds of changes, e.g., fixes of particular bug patterns, how to apply this kind of change to other code for automated program repair [10, 219, 239]. Another line of work ranks potential program repairs based on their similarity to common code change patterns [132]. DiffSearch could help gather datasets of changes for these approaches to learn from, e.g., based on queries for bug fixing patterns.

Moreover, the feature extractor of DiffSearch relates to techniques for learning vector representations of commits, such as CC2Vec [98] and Commit2Vec [27]. These techniques train a model on some “pseudo task” for which abundant training data is easily available, e.g., predicting the words in the commit message [98] or whether a commit is labeled as security-critical [27]. Once trained, the vector representations produced by a representation learning model could, in principle, be used as an alternative to the feature vectors of DiffSearch. In practice, integrating CC2Vec and Commit2Vec into our approach is non-trivial because both approaches focus on entire commits, which may include many hunks distributed across multiple files, whereas DiffSearch retrieves code changes at hunk-level granularity. Finding an appropriate pseudo task for representation learning on individual hunks, and integrating the resulting embeddings into DiffSearch, could be interesting future work.

6.5 Clone Detection

DiffSearch(Chapter 3) relates to code clone detectors [107, 114, 144, 220, 224], as answering a query resembles finding clones of the query. In particular, DiffSearch compares a query against code changes in a way similar to Type-1 clones, and when using placeholders in the query, similar to Type-2 and Type-3 clones. Clone detectors are typically evaluated on a single snapshot of a code base, and they may take several minutes or even hours to terminate [224]. In principle, one could use an off-the-shelf code clone detector to search for specific kinds of code changes, where the old and new parts of the query must be clones of the old and new parts of a change, respectively. However, this approach would search for clones among all code changes for each query, which may not be fast enough for an interactive search engine. Some clone detectors summarize code in ways related to our feature extraction. For example, Deckard [107] computes characteristic vectors of parse trees and SourcererCC [224] indexes large amounts of code into a bag-of-tokens representation. Integrating such ideas into the feature-based retrieval in DiffSearch could further improve recall.

Inoue et al. [103] propose a code clone detector that supports special tokens, such as $, $*$, #, to express exact matching, repetitions, and more, similar to regular expressions. However, their approach cannot express relationships between an old and a new code snippet, as supported by DiffSearch. Nguyen et al. [183] perform an empirical study on a large dataset of Java and C# using API2VEC based on Word2Vec to create feature vectors from APIs. They find this kind of representation successful, because APIs with similar usage context have closer feature vectors using this representation. DiffSearch differs from their approach because match code changes and because perform a matching based on the syntax of the code more than their usage context.

6.6 Type Annotations and Type Errors

Some prior work studies type annotations. For example, Khan et al. investigate the impact of using type checkers in Python [121], Rak-amnouykit et al. compare different type checkers with each other [211], and Zhang et al. provide evidence that static typing may reduce the bug fixing effort [285]. Both our study (Chapter 4) and Rak-amnouykit et al. [211] report that type-annotated repositories rarely type-check, showing the need for an APR tool for Python type errors, such as PyTy.

Jin et al. [110] study type annotations in 17 Python projects. They find six patterns that type annotations practices follow and they find three features of Python type annotation files. Chapter 4 differs from their result, because we analyze many more projects and we focus on different kinds of research questions focusing more on single type annotations and type errors.

Khan et al. [121] study type errors in Python repositories using *mypy*. They conclude that many type defects can be avoided (15%) by simply integrating a type checker in the software development process. Then, they find that junior and senior Python developers make a similar number of errors, concluding that the experience is not always enough to avoid this kind of errors. Chapter 4 differs from this one, because we not only analyze the number of type errors, but we study the relationship and the evolution between type annotations and type errors. Moreover, our study focuses on type annotations and not only type errors.

Researchers also study type annotations in programming languages other than Python, e.g., relating static type checking in JavaScript with known bugs [65]. Bogner and Merkel [21] compare JavaScript and TypeScript focusing on code quality and readability. However, also as our findings show (Chapter 4), they find that TypeScript does not always guarantee fewer errors. Finally, several studies of dynamically typed languages focus on questions complementary to ours, e.g., the use of dynamic language constructs [218], performance issues [228], and security vulnerabilities [240].

Gradual type checkers [188, 257] have developed into powerful tools for dynamically typed languages. Chen et al. build a framework to check type bugs, extracting information from source code using static analysis [32]. Dolby et al. use static analysis with types to track TensorFlow behavior and find bugs [47]. The results of our study (Chapter 4) underline the need for better integrating such tools into the development workflow.

6.7 Type Prediction for Dynamically Typed Languages

Techniques for predicting type annotations in dynamically typed languages fall into three categories. First, static type inference [7, 63, 91, 106] computes types using, e.g., abstract interpretation or type constraint propagation. While sound by design, these approaches are limited by the dynamic nature of the languages like JavaScript and Python. Second, dynamic type inference [6, 216] tracks data flows during an execution of a program, which yields precise types but is limited by code coverage. Third, probabilistic type prediction propagates and combines type hints using probabilistic rules [278] or via deep learning [4, 92, 157, 171, 213], sometimes augmented with search-based validation of predicted types [203] or static type inference [197]. Chapter 4 underlines the need for such techniques and PyTy (Chapter 5) addresses the complementary problem of fixing type-related errors.

Beyond type prediction, several other analyses for Python have been proposed, including techniques to find type-related bugs [277], an analysis to reveal inconsistencies between the name of a variable and the runtime values stored in it [194], and a general-purpose dynamic analysis framework [51]. These analyses are all based on dynamic analysis, which is at least partially motivated by the lack or incompleteness of type annotations in Python.

Earlier APR approaches [75] can be classified into heuristic repair, e.g., based on generate-and-validate method [123, 133], and constraint-based repair, which synthesizes a patch based constraints [166, 180]. Both techniques rely on test suites, and hence, may suffer from overfitting [206]. Chapter 5 belongs to a more recent stream of work on learning-based repair. In contrast to the above techniques, PyTy does not require tests but uses a static type checker to validate candidate fixes.

Other learning-based APR approaches include DrRepair [281], which fixes C compilation errors, Hoppity [45], which represents fixes as a sequence of graph edits, Recoder [288], based on TreeGen [247], which proposes a syntax-guided edit decoder. Compared to these GNN-based models, our approach uses a text-to-text transformer, which is easy to apply to any language. Other text-based models include SequenceR [35] and work by Tufano et al. [253]. Vasic et al. [254] propose to jointly localize and repair bugs, which is limited to variable-misuse bugs though. These approaches neither benefit from pre-training nor target Python type errors. CoCoNut [155] combines multiple models using ensemble learning. Instead, our approach in Chapter 5 learns how to fix all error types in one model. Ye et al. incorporate feedback from compiling and executing tests to train a repair model [283], an idea that could also be adapted to type error repair. Finally, motivated by recent results that show general-purpose LLMs to provide competitive results [108, 272, 273, 274], we empirically compare PyTy with different LLMs (Section 5.6.4).

We are aware of two APR approaches that target type errors. Rite [225] is a template-based, data-driven approach for type errors in OCaml.

Their approach builds on a specifically designed, AST-based representation of fixes, while our approach uses textual inputs and outputs.

PyTER [186] is a test-based APR approach to fix runtime type errors in Python, which we empirically compare with in Section 5.6.4. Their work and ours (Chapter 5) address related but ultimately different problems: PyTER requires test cases that trigger a runtime type error, but tests may not exist at all or have low coverage (e.g., Gruber et al. [78] report a median coverage of 3.7% across 22k Python projects). In contrast, PyTy addresses statically detectable errors, and hence, is limited to errors that are statically detectable. In practice, we expect PyTER and PyTy to complement each other.

Beyond type errors, several techniques for fixing other kinds of static analysis warnings have been proposed [10, 57, 158], which are also complementary to our work in Chapter 5. Recently, many pre-trained language models for source code have been proposed and achieve promising results [258], including CodeBERT [59], GraphCodeBERT [83], CodeT5 [263] and CodeTrans [55]. These models are pre-trained on large datasets and then fine-tuned on different tasks. We base our APR tool (Chapter 5) on TFix because it is already trained on an APR task. Another work that applies transfer learning in language models of code is VRepair [36]. They pre-train a transformer model on a large bug fix dataset for C, and then fine-tune it with a vulnerability fix dataset for C. Chapter 5 shows that the benefits of transferring knowledge are not only between different fixing tasks, but also between different programming languages (from JavaScript to Python).

Chapter 7
Conclusions and Future Work

Software evolution involves the growth and adaptation of software, including bug fixes, security patches, new programming languages features, and user-driven improvements. Effectively understanding and managing software evolution is crucial for ensuring sustained functionality and reliability in the field of software engineering. We focus on challenges that include the need for better information retrieval methods for software evolution (C-1), understanding developers' code changes patterns (C-2), and automating code changes for bug fixes (C-3). This dissertation argues that we can address these challenges with a mix of program analysis, information retrieval, and deep learning. Our contributions impact developers and researchers, with a comprehensive survey on code search guiding researchers on the state of the art (Chapter 2), DiffSearch offering a fast and scalable approach to search for code changes (Chapter 3), a large scale study on the evolution of type annotations in Python with insights for the developers and researchers (Chapter 4) and PyTy surpassing state-of-the-art techniques for automated program repair of Python type errors (Chapter 5). Collectively, these contributions advance understanding and practical capabilities in the field of software evolution.

7.1 Reflections and Lessons

During the work on this dissertation, we engaged in reflections, discussing valuable insights for the software engineering community. Here is a comprehensive summary of the lessons we learned:

- **Tools Adoption from Developers is Challenging.** One notable lesson is the challenges associated with developers adopting new tools. In the context of code search, the adoption of novel tools, such as those discussed in Chapter 2, can encounter resistance. In fact, even if there are so many code search approaches, most of the developers use only simple text to text code search. Bridging the gap between tool functionality and user adoption requires careful consideration of usability, integration, and user experience.
- **Code Changes are Repetitive.** The observation that code changes are repetitive (C-2) underlines the importance of approaches to find and reuse these patterns. In the realm of APR (C-3), recognizing and learning from repetitive patterns in code changes, as highlighted by PyTy, can significantly contribute to optimizing development workflows and reducing the manual workload.
- **The Usage of New Programming Language Features is Slow.** Acknowledging the challenges in adopting new programming language features is linked with the need to understand developers' patterns code changes (C-2). Integrating insights from more empirical studies on code changes can help in understanding how developers navigate and adapt to evolving language features, helping them with more effective support and guidelines.
- **Transfer Learning is Effective.** Another lesson from our journey is about the effectiveness of transfer learning. Leveraging transfer learning and a pre-trained model, as discussed in the context of PyTy, showcases its potential in improving automated program repair techniques without training a model from scratch and collecting a huge dataset.

- **Rule-based and Data-driven Approaches Can Work Together.** The synergy between rule-based and data-driven approaches aligns with the goal of addressing challenges in software evolution. Recognizing the compatibility of approaches like program analysis and deep learning, as advocated for PyTy where we use a type checker using traditional static analysis and a model based on T5, shows possibilities for hybrid solutions that can solve more software evolution challenges.
- **LLMs are Not the Solution to Everything.** While Large Language Models (LLMs) demonstrate remarkable capabilities, they still have limitations. As highlighted in Chapter 5, where PyTy performs better than GPT-4. LLMs are a valuable tool in the developer's and researcher's arsenal but may not be a universal solution, reinforcing the importance of hybrid solutions. However, the LLM potential is huge and researchers should continue to study them to improve the field of software evolution.

7.2 Research Vision and Future Work

Even if our last lesson in the previous section emphasizes the limitations of LLMs, they have opened new possibilities for code generation and automatic program repair in software development. However, the automatically generated code often needs to be carefully tested to ensure correctness and adherence to coding standards. Our future work will focus on addressing the challenges and opportunities associated with automatic program repair, bug detections and testing code generated by large language models using hybrid approaches also based on rule-based approaches. Some future work ideas are:

- **Automatic Program Repair with LLMs.** We can focus on AI prompt engineering to fix linter errors, for example detected by ErrorProne,[59] using LLMs with fine-tuning and few-shots prompting. The few-shot prompting can use DiffSearch to retrieve old useful fixes and help the model to provide a better fix.

[59] `https://errorprone.info/`

- **Bug Detection with LLMs:** As part of future work, we can explore the potential of LLMs for bug detection presents an interesting opportunity. For example, we can start from standard datasets such as Defect4J [112] and investigate how LLMs can effectively identify bugs. Additionally, fine-tuning existing LLMs for bug detection could contribute to the evolution of more robust and context-aware bug identification systems.
- **Automated Testing Techniques.** We can explore automated testing techniques for concurrent software systems [20] that leverage LLMs and program analysis to verify the correctness of tests generated by LLMs. This will involve developing novel testing approaches that can efficiently produce oracles and validate specific test cases and requirements.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007 (cit. on p. 15).

[2] S. Akbar, A. Kak. 'SCOR: Source Code Retrieval with Semantics and Order'. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 1–12 (cit. on pp. 20, 44).

[3] A. Alali, H. H. Kagdi, J. I. Maletic. 'What's a Typical Commit? A Characterization of Open Source Software Repositories'. In: *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*. Ed. by R. L. Krikhaar, R. Lammel, C. Verhoef. IEEE Computer Society, 2008, pp. 182–191. URL: https://doi.org/10.1109/ICPC.2008.24 (cit. on p. 60).

[4] M. Allamanis, E. T. Barr, S. Ducousso, Z. Gao. 'Typilus: Neural type hints'. In: *Proceedings of the 41st acm sigplan conference on programming language design and implementation*. 2020, pp. 91–105 (cit. on pp. 90, 116, 117, 120, 156).

[5] M. Allamanis, M. Brockschmidt, M. Khademi. 'Learning to Represent Programs with Graphs'. In: *International Conference on Learning Representations (ICLR)*. 2018 (cit. on p. 135).

[6] J.-h. (An, A. Chaudhuri, J. S. Foster, M. Hicks. 'Dynamic Inference of Static Types for Ruby'. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 459–472. URL: http://doi.acm.org/10.1145/1926385.1926437 (cit. on pp. 90, 156).

[7] C. Anderson, P. Giannini, S. Drossopoulou. 'Towards Type Inference for JavaScript'. In: *ECOOP 2005 - Object-Oriented Programming*. Ed. by A. P. Black. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 428–452 (cit. on pp. 90, 156).

[8] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh. 'Using Static Analysis to Find Bugs'. In: *IEEE Software* 25.5 (2008), pp. 22–29 (cit. on p. 4).

[9] A. Babenko, V. Lempitsky. ‘Efficient indexing of billion-scale datasets of deep descriptors’. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2055–2063 (cit. on p. 32).

[10] J. Bader, A. Scott, M. Pradel, S. Chandra. ‘Getafix: Learning to fix bugs automatically’. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–27 (cit. on pp. 56, 83, 153, 158).

[11] S. Bajracharya, T. Ngo, E. Linstead, P. Rigor, Y. Dou, P. Baldi, C. Lopes. ‘Sourcerer: a search engine for open source code’. In: *International Conference on Software Engineering (ICSE 2007)*. Citeseer. 2007 (cit. on pp. 34, 44, 58).

[12] S. K. Bajracharya, J. Ossher, C. V. Lopes. ‘Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories’. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE ’10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 157–166. URL: https://doi.org/10.1145/1882291.1882316 (cit. on pp. 20, 33, 37, 39, 44, 45).

[13] S. K. Bajracharya, C. V. Lopes. ‘Analyzing and mining a code search engine usage log’. In: *Empirical Software Engineering* 17.4-5 (2012), pp. 424–466. URL: https://doi.org/10.1007/s10664-010-9144-6 (cit. on pp. 47–50).

[14] S. K. Bajracharya, C. V. Lopes. ‘Mining search topics from a code search engine usage log’. In: *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*. Ed. by M. W. Godfrey, J. Whitehead. IEEE Computer Society, 2009, pp. 111–120. URL: https://doi.org/10.1109/MSR.2009.5069489 (cit. on pp. 33, 39).

[15] V. Balachandran. ‘Query by example in large-scale code repositories’. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 467–476 (cit. on pp. 22, 33, 34, 37, 39, 44, 45).

[16] M. Barnett, C. Bird, J. Brunet, S. K. Lahiri. ‘Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets’. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Ed. by A. Bertolino, G. Canfora, S. G. Elbaum. IEEE Computer Society, 2015, pp. 134–144. URL: https://doi.org/10.1109/ICSE.2015.35 (cit. on p. 60).

[17] R. Bavishi, H. Yoshida, M. R. Prasad. ‘Phoenix: Automated data-driven synthesis of repairs for static analysis violations’. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 613–624 (cit. on p. 152).

[18] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger. 'The R*-tree: An efficient and robust access method for points and rectangles'. In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 1990, pp. 322–331 (cit. on pp. 33, 39).

[19] B. Berabi, J. He, V. Raychev, M. Vechev. 'Tfix: Learning to fix coding errors with a text-to-text transformer'. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 780–791 (cit. on pp. 123, 131, 135, 138).

[20] F. A. Bianchi, A. Margara, M. Pezzè. 'A survey of recent trends in testing concurrent software systems'. In: *IEEE Transactions on Software Engineering* 44.8 (2017), pp. 747–783 (cit. on p. 162).

[21] J. Bogner, M. Merkel. 'To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github'. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 658–669 (cit. on p. 155).

[22] J. Brandt, M. Dontcheva, M. Weskamp, S. R. Klemmer. 'Example-centric programming: integrating web search into the development environment'. In: *Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, Georgia, USA, April 10-15, 2010*. Ed. by E. D. Mynatt, D. Schoner, G. Fitzpatrick, S. E. Hudson, W. K. Edwards, T. Rodden. ACM, 2010, pp. 513–522. URL: `https://doi.org/10.1145/1753326.1753402` (cit. on pp. 23, 53).

[23] S. Brin, L. Page. 'The anatomy of a large-scale hypertextual web search engine'. In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117 (cit. on p. 31).

[24] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei. 'Language Models are Few-Shot Learners'. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin. 2020. URL: `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html` (cit. on p. 52).

[25] M. Bruch, M. Monperrus, M. Mezini. 'Learning from examples to improve code completion systems'. In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, pp. 213–222 (cit. on p. 17).

[26] R. P. L. Buse, W. Weimer. 'Automatically documenting program changes'. In: *Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 33–42 (cit. on p. 151).

[27] R. Cabrera Lozoya, A. Baumann, A. Sabetta, M. Bezzi. 'Commit2vec: Learning distributed representations of code changes'. In: *SN Computer Science* 2.3 (2021), pp. 1–16 (cit. on pp. 87, 153).

[28] J. Cambronero, H. Li, S. Kim, K. Sen, S. Chandra. 'When deep learning met code search'. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by M. Dumas, D. Pfahl, S. Apel, A. Russo. ACM, 2019, pp. 964–974. URL: https://doi.org/10.1145/3338906.3340458 (cit. on pp. 33, 36, 40, 45).

[29] Y. Chai, H. Zhang, B. Shen, X. Gu. 'Cross-Domain Deep Code Search with Meta Learning'. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 487–498. URL: https://doi.org/10.1145/3510003.3510125 (cit. on pp. 20, 33–35, 41, 44).

[30] S. Chatterjee, S. Juvekar, K. Sen. 'Sniff: A search engine for java using free-form queries'. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2009, pp. 385–400 (cit. on pp. 20, 21, 44, 46).

[31] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, A. Michail. 'CVSSearch: Searching through source code using CVS comments'. In: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. IEEE. 2001, pp. 364–373 (cit. on pp. 20, 21, 33, 35, 36, 44, 45).

[32] L. Chen, B. Xu, T. Zhou, X. Zhou. 'A Constraint Based Bug Checking Approach for Python'. In: *Computer Software and Applications Conference (COMPSAC)*. IEEE, 2009, pp. 306–311 (cit. on p. 156).

[33] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, W. Zaremba. 'Evaluating Large Language Models Trained on Code'. In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374 (cit. on pp. 23, 51, 54).

[34] Q. Chen, M. Zhou. ‘A Neural Framework for Retrieval and Summarization of Source Code’. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: Association for Computing Machinery, 2018, pp. 826–831. URL: https://doi.org/10.1145/3238147.3240471 (cit. on pp. 20, 33, 36, 40, 44, 45).

[35] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, M. Monperrus. ‘SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair’. In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 1943–1959 (cit. on p. 157).

[36] Z. Chen, S. J. Kommrusch, M. Monperrus. ‘Neural Transfer Learning for Repairing Security Vulnerabilities in C Code’. In: *IEEE Transactions on Software Engineering* 49.1 (2022), pp. 147–165 (cit. on p. 158).

[37] Y. W. Chow, L. Di Grazia, M. Pradel. ‘PyTy: Repairing Static Type Errors in Python’. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society (cit. on p. 10).

[38] J. Cohen. ‘A coefficient of agreement for nominal scales’. In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46 (cit. on p. 139).

[39] B. Dagenais, M. P. Robillard. ‘Recommending Adaptive Changes for Framework Evolution’. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.4 (2011), pp. 1–35 (cit. on p. 152).

[40] Y. David, E. Yahav. ‘Tracelet-Based Code Search in Executables’. In: *Acm Sigplan Notices* 49.6 (2014), pp. 349–360 (cit. on pp. 23, 33, 34, 37, 42, 44, 45).

[41] L. Di Grazia, P. Bredl, M. Pradel. ‘DiffSearch: A Scalable and Precise Search Engine for Code Changes’. In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 2366–2380 (cit. on pp. 10, 50, 79).

[42] L. Di Grazia, M. Pradel. ‘Code Search: A Survey of Techniques for Finding Code’. In: *ACM Computing Surveys* (2022). URL: https://doi.org/10.1145/3565971 (cit. on pp. 10, 62).

[43] L. Di Grazia, M. Pradel. ‘The Evolution of Type Annotations in Python: An Empirical Study’. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 209–220. URL: https://doi.org/10.1145/3540250.3549114 (cit. on pp. 10, 120, 124, 125, 148, 149).

[44] T. Diamantopoulos, G. Karagiannopoulos, A. L. Symeonidis. 'Codecatch: Extracting Source Code Snippets from Online Sources'. In: *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. RAISE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 21–27. URL: https://doi.org/10.1145/3194104.3194107 (cit. on pp. 20, 33, 34, 36, 39, 44).

[45] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, K. Wang. 'Hoppity: Learning graph transformations to detect and fix bugs in programs'. In: *International Conference on Learning Representations (ICLR)*. 2020 (cit. on pp. 135, 157).

[46] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk. 'Feature location in source code: a taxonomy and survey'. In: *Journal of software: Evolution and Process* 25.1 (2013), pp. 53–95 (cit. on p. 18).

[47] J. Dolby, A. Shinnar, A. Allain, J. Reinen. 'Ariadne: analysis for machine learning programs'. In: *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2018, pp. 1–10 (cit. on p. 156).

[48] L. Du, X. Shi, Y. Wang, E. Shi, S. Han, D. Zhang. 'Is a Single Model Enough? MuCoS: A Multi-Model Ensemble Learning Approach for Semantic Code Search'. In: *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*. Ed. by G. Demartini, G. Zuccon, J. S. Culpepper, Z. Huang, H. Tong. ACM, 2021, pp. 2994–2998. URL: https://doi.org/10.1145/3459637.3482127 (cit. on pp. 20, 40).

[49] F. A. Durão, T. A. Vanderlei, E. S. Almeida, S. R. de L. Meira. 'Applying a Semantic Layer in a Source Code Search Tool'. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. SAC '08. Fortaleza, Ceara, Brazil: Association for Computing Machinery, 2008, pp. 1151–1157. URL: https://doi.org/10.1145/1363686.1363952 (cit. on p. 20).

[50] ECMA. *Standard ECMA-262, ECMAScript Language Specification, 5.1 Edition*. European Computer Manufacturers Association (ECMA), 2011 (cit. on p. 15).

[51] A. Eghbali, M. Pradel. 'DynaPyt: A Dynamic Analysis Framework for Python'. In: *ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022 (cit. on p. 156).

[52] A. Eghbali, M. Pradel. 'No Strings Attached: An Empirical Study of String-related Software Bugs'. In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 956–967. URL: https://ieeexplore.ieee.org/document/9286132 (cit. on p. 56).

[53] B. Elasticsearch. 'Elasticsearch'. In: *software], version* 6.1 (2018) (cit. on p. 4).

[54] B. Elkarablieh, S. Khurshid, D. Vu, K. S. McKinley. ‘Starc: static analysis for efficient repair of complex data’. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2007, pp. 387–404 (cit. on p. 4).

[55] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, B. Rost. ‘CodeTrans: Towards Cracking the Language of Silicon’s Code Through Self-Supervised Deep Learning and High Performance Computing’. In: *arXiv preprint arXiv:2104.02443* (2021) (cit. on p. 158).

[56] S. Erdweg, T. Szabó, A. Pacak. ‘Concise, Type-Safe, and Efficient Structural Diffing’. In: *PLDI*. 2021 (cit. on pp. 58, 152).

[57] K. Etemadi, N. Harrand, S. Larsén, H. Adzemovic, H. L. Phu, A. Verma, F. Madeiral, D. Wikström, M. Monperrus. ‘Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations’. In: *IEEE Transactions on Dependable and Secure Computing* 20.4 (2023), pp. 2794–2810 (cit. on p. 158).

[58] J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus. ‘Fine-grained and accurate source code differencing’. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*. 2014, pp. 313–324. URL: https://doi.org/10.1145/2642937.2642982 (cit. on pp. 58, 152).

[59] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou. ‘CodeBERT: A Pre-Trained Model for Programming and Natural Languages’. In: *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*. Ed. by T. Cohn, Y. He, Y. Liu. Vol. EMNLP 2020. Findings of ACL. Association for Computational Linguistics, 2020, pp. 1536–1547. URL: https://doi.org/10.18653/v1/2020.findings-emnlp.139 (cit. on pp. 41, 54, 158).

[60] C. Finn, P. Abbeel, S. Levine. ‘Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks’. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by D. Precup, Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1126–1135. URL: https://proceedings.mlr.press/v70/finn17a.html (cit. on p. 41).

[61] B. Fluri, M. Wuersch, M. PInzger, H. Gall. ‘Change distilling: Tree differencing for fine-grained source code change extraction’. In: *IEEE Transactions on software engineering* 33.11 (2007), pp. 725–743 (cit. on pp. 57, 58, 66, 152).

[62] Y. Fujiwara, N. Yoshida, E. Choi, K. Inoue. ‘Code-to-Code Search Based on Deep Neural Network and Code Mutation’. In: *2019 IEEE 13th International Workshop on Software Clones (IWSC)*. 2019, pp. 1–7 (cit. on pp. 22, 33, 34, 44).

[63] M. Furr, J. (An, J. S. Foster. 'Profile-guided static typing for dynamic scripting languages'. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2009, pp. 283–300 (cit. on pp. 90, 156).

[64] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, A. Tiwari. 'Feedback-driven semi-supervised synthesis of program transformations'. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 219:1–219:30. URL: `https://doi.org/10.1145/3428287` (cit. on p. 58).

[65] Z. Gao, C. Bird, E. T. Barr. 'To type or not to type: Quantifying detectable bugs in JavaScript'. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 758–769 (cit. on pp. 92, 114, 115, 155).

[66] V. C. Garcia, E. S. de Almeida, L. B. Lisboa, A. C. Martins, S. R. L. Meira, D. Lucredio, R. P. d. M. Fortes. 'Toward a Code Search Engine Based on the State-of-Art and Practice'. In: *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*. 2006, pp. 61–70 (cit. on p. 18).

[67] E. Giger, M. Pinzger, H. C. Gall. 'Comparing fine-grained source code changes and code churn for bug prediction'. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. 2011, pp. 83–92 (cit. on pp. 151, 152).

[68] GitHub. *The 2020 State of the Octoverse*. `https://octoverse.github.com/`. 2021 (cit. on p. 14).

[69] B. G. Glaser, A. L. Strauss, E. Strutzel. 'The discovery of grounded theory; strategies for qualitative research'. In: *Nursing research* 17.4 (1968), p. 364 (cit. on p. 127).

[70] M. W. Godfrey, D. M. German. 'The past, present, and future of software evolution'. In: *2008 Frontiers of Software Maintenance*. IEEE. 2008, pp. 129–138 (cit. on p. 1).

[71] I. Goodfellow, Y. Bengio, A. Courville. *Deep learning*. MIT press, 2016 (cit. on p. 4).

[72] A. Gosain, G. Sharma. 'Static analysis: A survey of techniques and tools'. In: *Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014*. Springer. 2015, pp. 581–591 (cit. on p. 4).

[73] J. Gosling, W. N. Joy, G. L. S. Jr. *The Java Language Specification*. Addison-Wesley, 1996 (cit. on p. 15).

[74] O. Gospodnetic, E. Hatcher, M. McCandless. *Lucene in action*. Simon and Schuster, 2010 (cit. on p. 4).

[75] C. L. Goues, M. Pradel, A. Roychoudhury. 'Automated program repair'. In: *Communications of the ACM* 62.12 (2019), pp. 56–65 (cit. on p. 157).

[76] L. D. Grazia, M. Pradel. *sola-st/PythonTypeAnnotationStudy: v1.0*. Version v1.0. Sept. 2022. URL: https://doi.org/10.5281/zenodo.7082252 (cit. on p. 94).

[77] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi-Reghizzi, D. Poshyvanyk, C. Fu, Q. Xie, C. Ghezzi. 'An empirical investigation into a large-scale Java open source code repository'. In: *Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2010 (cit. on p. 17).

[78] M. Gruber, S. Lukasczyk, F. Kroiß, G. Fraser. 'An empirical study of flaky tests in python'. In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2021, pp. 148–158 (cit. on p. 157).

[79] F. Grund, S. A. Chowdhury, N. Bradley, B. Hall, R. Holmes. 'CodeShovel: Constructing Method-Level Source Code Histories'. In: *ICSE*. 2021 (cit. on pp. 151, 153).

[80] J. Gu, Z. Chen, M. Monperrus. 'Multimodal Representation for Neural Code Search'. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 2021, pp. 483–494. URL: https://doi.org/10.1109/ICSME52107.2021.00049 (cit. on pp. 20, 41).

[81] R. Gu, G. Jin, L. Song, L. Zhu, S. Lu. 'What change history tells us about thread synchronization'. In: *ESEC/FSE*. 2015, pp. 426–438 (cit. on p. 152).

[82] X. Gu, H. Zhang, S. Kim. 'Deep Code Search'. In: *ICSE*. 2018 (cit. on pp. 20, 33, 35, 37, 40, 44, 52, 58).

[83] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, M. 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. URL: https://openreview.net/forum?id=jLoC4ez43PZ (cit. on pp. 41, 54, 158).

[84] T. Gvero, V. Kuncak. 'Synthesizing Java expressions from free-form queries'. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by J. Aldrich, P. Eugster. ACM, 2015, pp. 416–432. URL: https://doi.org/10.1145/2814270.2814295 (cit. on p. 17).

[85] A. Habib, M. Pradel. 'How Many of All Bugs Do We Find? A Study of Static Bug Detectors'. In: *ASE*. 2018 (cit. on pp. 56, 83).

[86] E. Hajiyev, M. Verbaere, O. de Moor. ‘codeQuest: Scalable Source Code Queries with Datalog’. In: *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*. Ed. by D. Thomas. Vol. 4067. Lecture Notes in Computer Science. Springer, 2006, pp. 2–27. URL: https://doi.org/10.1007/11785477%5C_2 (cit. on pp. 24, 33, 34, 37, 42).

[87] S. Hanenberg. ‘An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time’. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 2010, pp. 22–35 (cit. on pp. 92, 115).

[88] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, A. Stefik. ‘An empirical study on the impact of static typing on software maintainability’. In: *Empirical Software Engineering* 19.5 (2014), pp. 1335–1382 (cit. on p. 92).

[89] M. Hashimoto, A. Mori. ‘Diff/TS: A Tool for Fine-Grained Structural Change Analysis’. In: *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*. 2008, pp. 279–288. URL: https://doi.org/10.1109/WCRE.2008.44 (cit. on p. 152).

[90] M. Hashimoto, A. Mori, T. Izumida. ‘Automated patch extraction via syntax- and semantics-aware Delta debugging on source code changes’. In: *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*. 2018, pp. 598–609. URL: https://doi.org/10.1145/3236024.3236047 (cit. on pp. 151, 152).

[91] M. Hassan, C. Urban, M. Eilers, P. Müller. ‘MaxSMT-Based Type Inference for Python 3’. In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 12–19 (cit. on pp. 90, 156).

[92] V. J. Hellendoorn, C. Bird, E. T. Barr, M. Allamanis. ‘Deep learning type inference’. In: *ESEC/FSE*. Ed. by G. T. Leavens, A. Garcia, C. S. Pasareanu. ACM, 2018, pp. 152–162. URL: https://doi.org/10.1145/3236024.3236051 (cit. on pp. 90, 94, 116, 117, 120, 156).

[93] S. Herfert, J. Patra, M. Pradel. ‘Automatically Reducing Tree-Structured Test Inputs’. In: *ASE*. 2017 (cit. on p. 139).

[94] K. Herzig, S. Just, A. Zeller. ‘The impact of tangled code changes on defect prediction models’. In: *Empir. Softw. Eng.* 21.2 (2016), pp. 303–336. URL: https://doi.org/10.1007/s10664-015-9376-6 (cit. on p. 60).

[95] E. Hill, L. Pollock, K. Vijay-Shanker. ‘Automatically capturing source code context of NL-queries for software maintenance and reuse’. In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 232–242 (cit. on pp. 20, 21).

[96] E. Hill, L. Pollock, K. Vijay-Shanker. ‘Improving source code search with natural language phrasal representations of method signatures’. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 524–527 (cit. on pp. 20, 21).

[97] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, N. Ubayashi. ‘DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction’. In: *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.* 2019, pp. 34–45. URL: https://doi.org/10.1109/MSR.2019.00016 (cit. on pp. 151, 152).

[98] T. Hoang, H. J. Kang, D. Lo, J. Lawall. ‘Cc2vec: Distributed representations of code changes’. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 518–529 (cit. on pp. 87, 153).

[99] R. Holmes, G. C. Murphy. ‘Using Structural Context to Recommend Source Code Examples’. In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 117–125. URL: https://doi.org/10.1145/1062455.1062491 (cit. on pp. 23, 33, 34, 38, 44).

[100] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, N. Duan. ‘CoSQA: 20, 000+ Web Queries for Code Search and Question Answering’. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*. Ed. by C. Zong, F. Xia, W. Li, R. Navigli. Association for Computational Linguistics, 2021, pp. 5690–5700. URL: https://doi.org/10.18653/v1/2021.acl-long.442 (cit. on p. 53).

[101] H. Husain, H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt. ‘CodeSearchNet Challenge: Evaluating the State of Semantic Code Search’. In: *CoRR* abs/1909.09436 (2019). arXiv: 1909.09436. URL: http://arxiv.org/abs/1909.09436 (cit. on pp. 15, 33, 34, 40, 44, 53).

[102] H. Husain, H.-H. Wu. ‘How to create natural language semantic search for arbitrary objects with deep learning’. In: *Retrieved November* 5 (2018), p. 2019 (cit. on p. 40).

[103] K. Inoue, Y. Miyamoto, D. M. German, T. Ishio. ‘Code clone matching: A practical and effective approach to find code snippets’. In: *arXiv preprint arXiv:2003.05615* (2020) (cit. on pp. 14, 25, 33, 34, 36, 154).

[104] *ISO / IEC 14882 international standard - first edition 1998-09-01: Programming languages C++*. ISO, 1998 (cit. on p. 15).

[105] D. Janzen, K. D. Volder. ‘Navigating and querying code without getting lost’. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA, March 17-21, 2003*. Ed. by W. G. Griswold, M. Aksit. ACM, 2003, pp. 178–187. URL: https://doi.org/10.1145/643603.643622 (cit. on p. 24).

[106] S. H. Jensen, A. Møller, P. Thiemann. ‘Type Analysis for JavaScript’. In: *Symposium on Static Analysis (SAS)*. Springer, 2009, pp. 238–255 (cit. on pp. 90, 156).

[107] L. Jiang, G. Misherghi, Z. Su, S. Glondu. ‘Deckard: Scalable and accurate tree-based detection of code clones’. In: *29th International Conference on Software Engineering (ICSE’07)*. IEEE. 2007, pp. 96–105 (cit. on pp. 58, 154).

[108] N. Jiang, K. Liu, T. Lutellier, L. Tan. ‘Impact of code language models on automated program repair’. In: *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*. 2023 (cit. on pp. 146, 157).

[109] R. Jiang, Z. Chen, Z. Zhang, Y. Pei, M. Pan, T. Zhang. ‘[Research Paper] Semantics-Based Code Search Using Input/Output Examples’. In: *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. IEEE Computer Society, 2018, pp. 92–102. URL: https://doi.org/10.1109/SCAM.2018.00018 (cit. on pp. 26, 33, 35, 43, 44, 46).

[110] W. Jin, D. Zhong, Z. Ding, M. Fan, T. Liu. ‘Where to Start: Studying Type Annotation Practices in Python’. In: *ASE*. 2021 (cit. on p. 155).

[111] J. Johnson, M. Douze, H. Jégou. ‘Billion-scale similarity search with GPUs’. In: *IEEE Transactions on Big Data* (2019) (cit. on pp. 62, 68, 73, 76).

[112] R. Just, D. Jalali, M. D. Ernst. ‘Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs’. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440. URL: https://doi.org/10.1145/2610384.2628055 (cit. on p. 162).

[113] H. H. Kagdi, M. L. Collard, J. I. Maletic. ‘A survey and taxonomy of approaches for mining software repositories in the context of software evolution’. In: *J. Softw. Maintenance Res. Pract.* 19.2 (2007), pp. 77–131. URL: https://doi.org/10.1002/smr.344 (cit. on p. 17).

[114] T. Kamiya, S. Kusumoto, K. Inoue. ‘CCFinder: a multilinguistic token-based code clone detection system for large scale source code’. In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670 (cit. on pp. 58, 154).

[115] R.-M. Karampatsis, C. Sutton. 'How often do single-statement bugs occur? the manysstubs4j dataset'. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 573–577 (cit. on pp. 74, 83, 84).

[116] D. Kawrykow, M. P. Robillard. 'Non-essential changes in version histories'. In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE. 2011, pp. 351–360 (cit. on pp. 57, 60).

[117] Y. Ke, K. T. Stolee, C. Le Goues, Y. Brun. 'Repairing programs with semantic code search (t)'. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 295–306 (cit. on p. 26).

[118] J. D. M.-W. C. Kenton, L. K. Toutanova. 'Bert: Pre-training of deep bidirectional transformers for language understanding'. In: 1 (2019), p. 2 (cit. on p. 41).

[119] A. Ketkar, N. Tsantalis, D. Dig. 'Understanding type changes in java'. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 629–641 (cit. on p. 153).

[120] M. Khalifa. 'Semantic Source Code Search: A Study of the Past and a Glimpse at the Future'. In: *CoRR* abs/1908.06738 (2019). arXiv: 1908.06738. URL: http://arxiv.org/abs/1908.06738 (cit. on p. 18).

[121] F. Khan, B. Chen, D. Varro, S. Mcintosh. 'An Empirical Study of Type-Related Defects in Python Projects'. In: *IEEE Transactions on Software Engineering* (2021) (cit. on pp. 92, 114, 115, 155).

[122] W. M. Khoo, A. Mycroft, R. Anderson. 'Rendezvous: A search engine for binary code'. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013, pp. 329–338 (cit. on pp. 23, 33, 34, 36, 39).

[123] D. Kim, J. Nam, J. Song, S. Kim. 'Automatic patch generation learned from human-written patches.' In: *International Conference on Software Engineering (ICSE)*. 2013, pp. 802–811 (cit. on p. 157).

[124] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, Y. L. Traon. 'FaCoY: a code-to-code search engine'. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 946–957 (cit. on pp. 22, 23, 33, 37, 39, 44, 46, 58).

[125] S. Kim, E. J. W. Jr., Y. Zhang. 'Classifying Software Changes: Clean or Buggy?' In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 181–196 (cit. on p. 153).

[126] A. J. Ko, B. A. Myers, M. J. Coblenz, H. H. Aung. 'An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks'. In: *IEEE Trans. Software Eng.* 32.12 (2006), pp. 971–987. URL: https://doi.org/10.1109/TSE.2006.116 (cit. on pp. 47–49).

[127] T. Kudo, J. Richardson. ‘SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing’. In: *Conference on Empirical Methods in Natural Language Processing*. 2018 (cit. on p. 136).

[128] J. R. Landis, G. G. Koch. ‘The Measurement of Observer Agreement for Categorical Data’. In: *Biometrics* 33.1 (1977), pp. 159–174. URL: http://www.jstor.org/stable/2529310 (visited on 01/23/2023) (cit. on p. 139).

[129] J. Lawall, Q. Lambert, G. Muller. ‘Prequel: A patch-like query language for commit history search’. PhD thesis. Inria Paris, 2016 (cit. on pp. 25, 50).

[130] J. Lawall, G. Muller. ‘Coccinelle: 10 Years of Automated Evolution in the Linux Kernel’. In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by H. S. Gunawi, B. Reed. USENIX Association, 2018, pp. 601–614. URL: https://www.usenix.org/conference/atc18/presentation/lawall (cit. on pp. 4, 25, 73).

[131] J. Lawall, D. Palinski, L. Gnirke, G. Muller. ‘Fast and precise retrieval of forward and back porting information for Linux device drivers’. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 2017, pp. 15–26 (cit. on pp. 57, 66).

[132] X. D. Le, D. Lo, C. Le Goues. ‘History Driven Program Repair’. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. 2016, pp. 213–224. URL: https://doi.org/10.1109/SANER.2016.76 (cit. on p. 153).

[133] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer. ‘GenProg: A Generic Method for Automatic Software Repair’. In: *IEEE Trans. Software Eng.* 38.1 (2012), pp. 54–72 (cit. on p. 157).

[134] C. Le Goues, M. Pradel, A. Roychoudhury. ‘Automated program repair’. In: *Commun. ACM* 62.12 (2019), pp. 56–65. URL: https://doi.org/10.1145/3318162 (cit. on pp. 56, 122, 146).

[135] C. Leacock, M. Chodorow. ‘Combining local context and WordNet similarity for word sense identification’. In: *WordNet: An electronic lexical database* 49.2 (1998), pp. 265–283 (cit. on p. 30).

[136] K. Lee, M. Chang, K. Toutanova. ‘Latent Retrieval for Weakly Supervised Open Domain Question Answering’. In: *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. Ed. by A. Korhonen, D. R. Traum, L. Marquez. Association for Computational Linguistics, 2019, pp. 6086–6096. URL: https://doi.org/10.18653/v1/p19-1612 (cit. on p. 52).

[137] M. Lee, S. Hwang, S. Kim. ‘Integrating code search into the development session’. In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 1336–1339 (cit. on pp. 22, 33, 34, 45).

[138] M.-W. Lee, J.-W. Roh, S.-w. Hwang, S. Kim. 'Instant Code Clone Search'. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 167–176. URL: https://doi.org/10.1145/1882291.1882317 (cit. on pp. 22, 33, 34, 39, 45).

[139] M. Lehman, J. C. Fernáandez-Ramil. 'Software evolution'. In: *Software evolution and feedback: Theory and practice* (2006), pp. 7–40 (cit. on p. 1).

[140] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, C. V. Lopes. 'A test-driven approach to code search and its application to the reuse of auxiliary functionality'. In: *Inf. Softw. Technol.* 53.4 (2011), pp. 294–306. URL: https://doi.org/10.1016/j.infsof.2010.11.009 (cit. on pp. 26, 44).

[141] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, C. V. Lopes. 'CodeGenie: using test-cases to search and reuse source code'. In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. Ed. by R. E. K. Stirewalt, A. Egyed, B. Fischer. ACM, 2007, pp. 525–526. URL: https://doi.org/10.1145/1321631.1321726 (cit. on p. 26).

[142] W. Li, S. Yan, B. Shen, Y. Chen. 'Reinforcement Learning of Code Search Sessions'. In: *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*. IEEE, 2019, pp. 458–465. URL: https://doi.org/10.1109/APSEC48747.2019.00068 (cit. on pp. 28, 29).

[143] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, H. Mei. 'Relationship-Aware Code Search for JavaScript Frameworks'. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 690–701. URL: https://doi.org/10.1145/2950290.2950341 (cit. on pp. 20, 28, 30, 31, 33, 34, 38, 42, 44, 45).

[144] Z. Li, S. Lu, S. Myagmar, Y. Zhou. 'CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code'. In: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 176–192 (cit. on pp. 58, 154).

[145] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, S. Ji. 'Deep Graph Matching and Searching for Semantic Code Retrieval'. In: *ACM Trans. Knowl. Discov. Data* 15.5 (2021), 88:1–88:21. URL: https://doi.org/10.1145/3447571 (cit. on pp. 20, 34, 37, 41, 45).

[146] E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, P. Baldi. 'Sourcerer: mining and searching internet-scale software repositories'. In: *Data Min. Knowl. Discov.* 18.2 (2009), pp. 300–336. URL: https://doi.org/10.1007/s10618-008-0118-x (cit. on pp. 20, 33, 35, 37, 44).

[147] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, J. C. Grundy. 'Opportunities and Challenges in Code Search Tools'. In: *ACM Comput. Surv.* 54.9 (2022), 196:1–196:40. URL: https://doi.org/10.1145/3480027 (cit. on pp. 4, 18).

[148] K. Liu, D. Kim, T. F. Bissyande, T. Kim, K. Kim, A. Koyuncu, S. Kim, Y. L. Traon. 'Learning to spot and refactor inconsistent method names'. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, pp. 1–12. URL: https://dl.acm.org/citation.cfm?id=3339507 (cit. on p. 20).

[149] X. Liu, X. Kong, L. Liu, K. Chiang. 'TreeGAN: syntax-aware sequence generation with generative adversarial networks'. In: (2018), pp. 1140–1145 (cit. on p. 20).

[150] V. B. Livshits, T. Zimmermann. 'DynaMine: Finding common error patterns by mining software revision histories'. In: *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005, pp. 296–305 (cit. on p. 153).

[151] J. Lu, Y. Wei, X. Sun, B. Li, W. Wen, C. Zhou. 'Interactive Query Reformulation for Source-Code Search With Word Relations'. In: *IEEE Access* 6 (2018), pp. 75660–75668 (cit. on pp. 20, 28, 44, 46).

[152] M. Lu, X. Sun, S. Wang, D. Lo, Y. Duan. 'Query expansion via wordnet for effective code search'. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. 2015, pp. 545–549 (cit. on pp. 20, 21, 28, 30, 44, 45).

[153] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, S. Liu. 'CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation'. In: *CoRR*. Vol. abs/2102.04664. 2021 (cit. on p. 53).

[154] S. Luan, D. Yang, C. Barnaby, K. Sen, S. Chandra. 'Aroma: Code recommendation via structural code search'. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), p. 152 (cit. on pp. 22, 33, 34, 37, 39, 44, 47, 58).

[155] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, L. Tan. 'CoCoNuT: combining context-aware neural translation models using ensemble for program repair'. In: *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. Ed. by S. Khurshid, C. S. Pasareanu. ACM, 2020, pp. 101–114. URL: https://doi.org/10.1145/3395363.3397369 (cit. on p. 157).

[156] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, J. Zhao. 'CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E)'. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. 2015, pp. 260–270 (cit. on pp. 20, 28, 30, 31, 33, 37, 44).

[157] R. S. Malik, J. Patra, M. Pradel. 'NL2Type: Inferring JavaScript function types from natural language information'. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, pp. 304–315. URL: https://doi.org/10.1109/ICSE.2019.00045 (cit. on pp. 90, 94, 116, 117, 156).

[158] D. Marcilio, C. A. Furia, R. Bonifácio, G. Pinto. 'SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings'. In: *J. Syst. Softw.* 168 (2020), p. 110671. URL: https://doi.org/10.1016/j.jss.2020.110671 (cit. on p. 158).

[159] L. Martie, A. v. d. Hoek. 'Sameness: An Experiment in Code Search'. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 76–87 (cit. on pp. 20, 21).

[160] L. Martie, A. v. d. Hoek, T. Kwak. 'Understanding the Impact of Support for Iteration on Code Search'. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 774–785. URL: https://doi.org/10.1145/3106237.3106293 (cit. on pp. 27–29).

[161] L. Martie, T. D. LaToza, A. van der Hoek. 'Codeexchange: Supporting reformulation of internet-scale code queries in context (t)'. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 24–35 (cit. on pp. 25, 28, 29, 33, 39).

[162] G. Mathew, C. Parnin, K. T. Stolee. 'SLACC: simion-based language agnostic code clones'. In: *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. Ed. by G. Rothermel, D. Bae. ACM, 2020, pp. 210–221. URL: https://doi.org/10.1145/3377811.3380407 (cit. on p. 35).

[163] G. Mathew, K. T. Stolee. 'Cross-language code search using static and dynamic analyses'. In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. Ed. by D. Spinellis, G. Gousios, M. Chechik, M. D. Penta. ACM, 2021, pp. 205–217. URL: https://doi.org/10.1145/3468264.3468538 (cit. on pp. 22, 34, 35, 46).

[164] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, Q. Xie. ‘Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications’. In: *IEEE Transactions on Software Engineering* 38.5 (2012), pp. 1069–1087 (cit. on p. 17).

[165] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, C. Fu. ‘Portfolio: Searching for Relevant Functions and Their Usages in Millions of Lines of Code’. In: *ACM Trans. Softw. Eng. Methodol.* 22.4 (Oct. 2013). URL: https://doi.org/10.1145/2522920.2522930 (cit. on pp. 20, 21, 33, 42, 44).

[166] S. Mechtaev, J. Yi, A. Roychoudhury. ‘Angelix: Scalable multiline program patch synthesis via symbolic analysis’. In: *Proceedings of the 38th international conference on software engineering*. 2016, pp. 691–701 (cit. on p. 157).

[167] N. Meng, M. Kim, K. S. McKinley. ‘Systematic editing: generating program transformations from an example’. In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 329–342 (cit. on p. 152).

[168] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, M. Jazayeri. ‘Challenges in software evolution’. In: *Eighth International Workshop on Principles of Software Evolution (IWPSE’05)*. 2005, pp. 13–22 (cit. on p. 1).

[169] A. Michail. ‘Browsing and Searching Source Code of Applications Written Using a GUI Framework’. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE ’02. Orlando, Florida: Association for Computing Machinery, 2002, pp. 327–337. URL: https://doi.org/10.1145/581339.581381 (cit. on p. 20).

[170] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean. ‘Distributed representations of words and phrases and their compositionality’. In: *Advances in neural information processing systems*. 2013, pp. 3111–3119 (cit. on pp. 21, 30).

[171] A. M. Mir, E. Latoškinas, S. Proksch, G. Gousios. ‘Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python’. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2241–2252. URL: https://doi.org/10.1145/3510003.3510124 (cit. on p. 156).

[172] A. Mishne, S. Shoham, E. Yahav. ‘Typestate-Based Semantic Code Search over Partial Programs’. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 997–1016. URL: https://doi.org/10.1145/2384616.2384689 (cit. on pp. 22, 23, 33, 38, 44, 47).

[173] B. Mitra, N. Craswell, et al. *An introduction to neural information retrieval*. Now Foundations and Trends, 2018 (cit. on p. 40).

[174] M. Motwani, M. Soto, Y. Brun, R. Just, C. Le Goues. 'Quality of automated program repair on real-world defects'. In: *IEEE Transactions on Software Engineering* (2020) (cit. on p. 56).

[175] R. Mukherjee, S. Chaudhuri, C. Jermaine. 'Searching a Database of Source Codes Using Contextualized Code Search'. In: *Proc. VLDB Endow.* 13.10 (2020), pp. 1765–1778. URL: `https://doi.org/10.14778/3401960.3401972` (cit. on pp. 22, 23, 33, 41, 44).

[176] S. Negara, M. Codoban, D. Dig, R. E. Johnson. 'Mining fine-grained code changes to detect unknown change patterns'. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 803–813 (cit. on pp. 56, 153).

[177] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, D. 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*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 511–522. URL: `https://doi.org/10.1145/2950290.2950333` (cit. on pp. 22, 33, 44, 153).

[178] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, H. Rajan. 'A study of repetitiveness of code changes in software evolution'. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 180–190 (cit. on p. 153).

[179] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, M. Hilton. 'Graph-based mining of in-the-wild, fine-grained, semantic code change patterns'. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, pp. 819–830. URL: `https://doi.org/10.1109/ICSE.2019.00089` (cit. on pp. 56, 58, 153).

[180] H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra. 'SemFix: program repair via semantic analysis'. In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 772–781 (cit. on p. 157).

[181] K. Nguyen, G. Xu. 'Cachetor: Detecting Cacheable Data to Remove Bloat'. In: *European Software Engineering Conference and International Symposiumon Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 268–278 (cit. on p. 152).

[182] T. V. Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, T. N. Nguyen. 'Combining Word2Vec with Revised Vector Space Model for Better Code Retrieval'. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 183–185 (cit. on pp. 15, 20, 21, 33, 34, 39, 44).

[183] T. D. Nguyen, A. T. Nguyen, H. D. Phan, T. N. Nguyen. 'Exploring API embedding for API usages and applications'. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 438–449 (cit. on p. 154).

[184] L. Nie, H. Jiang, Z. Ren, Z. Sun, X. Li. 'Query Expansion Based on Crowd Knowledge for Code Search'. In: *IEEE Transactions on Services Computing* 9.5 (2016), pp. 771–783 (cit. on pp. 20, 28, 30, 33, 36, 44).

[185] B. B. Nielsen, M. T. Torp, A. Møller. 'Semantic patches for adaptation of JavaScript programs to evolving libraries'. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 74–85 (cit. on p. 151).

[186] W. Oh, H. Oh. 'PyTER: Effective Program Repair for Python Type Errors'. In: *ESEC/FSE*. 2022 (cit. on pp. 123, 147, 148, 157).

[187] J.-P. Ore, S. Elbaum, C. Detweiler, L. Karkazis. 'Assessing the type annotation burden'. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 190–201 (cit. on p. 92).

[188] F. Ortin, J. B. G. Perez-Schofield, J. M. Redondo. 'Towards a static type checker for python'. In: *European Conference on Object-Oriented Programming (ECOOP), Scripts to Programs Workshop, STOP*. Vol. 15. 2015, pp. 1–2 (cit. on pp. 90, 156).

[189] J. Ossher, S. K. Bajracharya, E. Linstead, P. Baldi, C. V. Lopes. 'SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects'. In: *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*. Ed. by M. W. Godfrey, J. Whitehead. IEEE Computer Society, 2009, pp. 183–186. URL: `https://doi.org/10.1109/MSR.2009.5069501` (cit. on pp. 33, 44).

[190] R. Paletov, P. Tsankov, V. Raychev, M. T. Vechev. 'Inferring crypto API rules from code changes'. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by J. S. Foster, D. Grossman. ACM, 2018, pp. 450–464. URL: `https://doi.org/10.1145/3192366.3192403` (cit. on pp. 58, 152, 153).

[191] K. Pan, S. Kim, E. J. Whitehead. 'Toward an understanding of bug fix patterns'. In: *Empirical Software Engineering* 14.3 (2009), pp. 286–315 (cit. on p. 57).

[192] O. Panchenko, H. Plattner, A. Zeier. 'What do developers search for in source code and why'. In: *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. 2011, pp. 33–36 (cit. on pp. 47, 48).

[193] P. P. Partachi, S. K. Dash, M. Allamanis, E. T. Barr. 'Flexeme: untangling commits using lexical flows'. In: *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*. Ed. by P. Devanbu, M. B. Cohen, T. Zimmermann. ACM, 2020, pp. 63–74. URL: `https://doi.org/10.1145/3368089.3409693` (cit. on p. 60).

[194] J. Patra, M. Pradel. 'Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks'. In: *ICSE*. 2022 (cit. on p. 156).

[195] S. Paul. 'SCRUPLE: A Reengineer's Tool for Source Code Search'. In: *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research - Volume 1*. CASCON '92. Toronto, Ontario, Canada: IBM Press, 1992, pp. 329–346 (cit. on pp. 22, 33, 34, 38, 43, 44).

[196] S. Paul, A. Prakash. 'A framework for source code search using program patterns'. In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 463–475 (cit. on pp. 22, 28, 31, 33, 34, 38, 43, 44).

[197] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, M. 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. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2019–2030. URL: `https://doi.org/10.1145/3510003.3510038` (cit. on p. 156).

[198] J. Pennington, R. Socher, C. D. Manning. 'Glove: Global vectors for word representation'. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543 (cit. on p. 41).

[199] J. A. Pienaar, R. Hundt. 'JSWhiz: Static analysis for JavaScript memory leaks'. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. 2013, 11:1–11:11 (cit. on p. 4).

[200] A. Podgurski, L. Pierce. 'Retrieving Reusable Software by Sampling Behaviour'. In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (1993), pp. 286–303. URL: `https://doi.org/10.1145/152388.152392` (cit. on pp. 26, 33, 35, 44).

[201] D. Poshyvanyk, M. Grechanik. 'Creating and evolving software by searching, selecting and synthesizing relevant source code'. In: *2009 31st International Conference on Software Engineering - Companion Volume*. 2009, pp. 283–286 (cit. on pp. 33, 39, 44).

[202] M. Pradel, S. Chandra. 'Neural software analysis'. In: *Commun. ACM* 65.1 (2022), pp. 86–96. URL: `https://doi.org/10.1145/3460348` (cit. on pp. 40, 52).

[203] M. Pradel, G. Gousios, J. Liu, S. Chandra. 'TypeWriter: Neural Type Prediction with Search-based Validation'. In: *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*. 2020, pp. 209–220. URL: https://doi.org/10.1145/3368089.3409715 (cit. on pp. 90, 94, 116, 117, 120, 156).

[204] M. Pradel, K. Sen. 'DeepBugs: A learning approach to name-based bug detection'. In: *PACMPL* 2.OOPSLA (2018), 147:1–147:25. URL: https://doi.org/10.1145/3276517 (cit. on pp. 83, 120).

[205] V. Premtoon, J. Koppel, A. Solar-Lezama. 'Semantic Code Search via Equational Reasoning'. In: *PLDI*. 2020 (cit. on pp. 26, 38, 42, 44).

[206] Z. Qi, F. Long, S. Achour, M. Rinard. 'An analysis of patch plausibility and correctness for generate-and-validate patch generation systems'. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 2015, pp. 24–36 (cit. on p. 157).

[207] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu. 'Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer'. In: *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. URL: http://jmlr.org/papers/v21/20-074.html (cit. on p. 135).

[208] M. Raghothaman, Y. Wei, Y. Hamadi. 'SWIM: Synthesizing What i Mean: Code Search and Idiomatic Snippet Synthesis'. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 357–367. URL: https://doi.org/10.1145/2884781.2884808 (cit. on pp. 20, 21).

[209] M. M. Rahman, C. Roy. 'Effective Reformulation of Query for Code Search Using Crowdsourced Knowledge and Extra-Large Data Analytics'. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 473–484 (cit. on pp. 20, 28, 30, 31, 46).

[210] M. M. Rahman, J. Barson, S. Paul, J. Kayan, F. A. Lois, S. F. Quezada, C. Parnin, K. T. Stolee, B. Ray. 'Evaluating how developers use general-purpose web-search for code retrieval'. In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by A. Zaidman, Y. Kamei, E. Hill. ACM, 2018, pp. 465–475. URL: https://doi.org/10.1145/3196398.3196425 (cit. on pp. 14, 47–50).

[211] I. Rak-amnouykit, D. McCrevan, A. Milanova, M. Hirzel, J. Dolby. 'Python 3 Types in the Wild: A Tale of Two Type Systems'. In: *DLS*. 2020 (cit. on pp. 56, 93, 116, 124, 155).

[212] N. Rao, C. Bansal, J. Guan. 'Search4Code: Code Search Intent Classification Using Weak Supervision'. In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 575–579. URL: https://doi.org/10.1109/MSR52588.2021.00077 (cit. on p. 53).

[213] V. Raychev, M. T. Vechev, A. Krause. 'Predicting Program Properties from "Big Code".' In: *Principles of Programming Languages (POPL)*. 2015, pp. 111–124 (cit. on pp. 90, 156).

[214] V. Raychev, M. T. Vechev, E. Yahav. 'Code completion with statistical language models'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 2014, p. 44 (cit. on p. 17).

[215] S. P. Reiss. 'Semantics-based code search'. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 243–253. URL: https://doi.org/10.1109/ICSE.2009.5070525 (cit. on pp. 26, 27, 35, 44, 58).

[216] B. M. Ren, J. Toman, T. S. Strickland, J. S. Foster. 'The Ruby Type Checker'. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1565–1572. URL: http://doi.acm.org/10.1145/2480362.2480655 (cit. on pp. 90, 156).

[217] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, Y. Arroyo-Paredes. 'Detecting Argument Selection Defects'. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2017 (cit. on p. 59).

[218] G. Richards, C. Hammer, B. Burg, J. Vitek. 'The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications'. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2011, pp. 52–78 (cit. on p. 155).

[219] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, B. Hartmann. 'Learning syntactic program transformations from examples'. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, pp. 404–415. URL: https://doi.org/10.1109/ICSE.2017.44 (cit. on pp. 58, 153).

[220] C. K. Roy, J. R. Cordy. 'NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization'. In: *2008 16th iEEE international conference on program comprehension*. IEEE. 2008, pp. 172–181 (cit. on pp. 58, 154).

[221] C. K. Roy, J. R. Cordy. 'A survey on software clone detection research'. In: *Queen's School of Computing TR* 541.115 (2007), pp. 64–68 (cit. on pp. 17, 51).

[222] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, S. Chandra. 'Retrieval on source code: a neural code search'. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM. 2018, pp. 31–41 (cit. on pp. 20, 33, 35, 40, 41, 44, 45).

[223] C. Sadowski, K. T. Stolee, S. Elbaum. 'How developers search for code: a case study'. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 191–201 (cit. on pp. 14, 32, 47–49, 78).

[224] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes. 'SourcererCC: Scaling code clone detection to big-code'. In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 1157–1168 (cit. on pp. 58, 154).

[225] G. Sakkas, M. Endres, B. Cosman, W. Weimer, R. Jhala. 'Type error feedback via analytic program repair'. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by A. F. Donaldson, E. Torlak. ACM, 2020, pp. 16–30. URL: `https://doi.org/10.1145/3385412.3386005` (cit. on p. 157).

[226] G. Salton, E. A. Fox, H. Wu. 'Extended Boolean information retrieval'. In: *Communications of the ACM* 26.11 (1983), pp. 1022–1036 (cit. on p. 39).

[227] P. Salza, C. Schwizer, J. Gu, H. C. Gall. 'On the effectiveness of transfer learning for code search'. In: *IEEE Transactions on Software Engineering* (2022) (cit. on pp. 20, 33, 34, 37, 41, 44, 45).

[228] M. Selakovic, M. Pradel. 'Performance Issues and Optimizations in JavaScript: An Empirical Study'. In: *International Conference on Software Engineering (ICSE)*. 2016, pp. 61–72 (cit. on p. 155).

[229] T. Seymour, D. Frantsvog, S. Kumar, et al. 'History of search engines'. In: *International Journal of Management & Information Systems (IJMIS)* 15.4 (2011), pp. 47–58 (cit. on p. 14).

[230] D. Shepherd, K. Damevski, B. Ropski, T. Fritz. 'Sando: An Extensible Local Code Search Framework'. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: Association for Computing Machinery, 2012. URL: `https://doi.org/10.1145/2393596.2393612` (cit. on pp. 15, 20, 21, 28, 31, 33, 34).

[231] J. G. Siek, W. Taha. 'Gradual Typing for Objects'. In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Ed. by E. Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 2–27. URL: `https://doi.org/10.1007/978-3-540-73589-2%5C_2` (cit. on p. 124).

[232] S. E. Sim, C. L. A. Clarke, R. C. Holt. ‘Archetypal Source Code Searches: A Survey of Software Developers and Maintainers’. In: *6th International Workshop on Program Comprehension (IWPC ’98), June 24-26, 1998, Ischia, Italy*. IEEE Computer Society, 1998, pp. 180–187. URL: https://doi.org/10.1109/WPC.1998.693351 (cit. on pp. 47–49).

[233] S. E. Sim, M. Umarji, S. Ratanotayanon, C. V. Lopes. ‘How Well Do Search Engines Support Code Retrieval on the Web?’ In: *ACM Trans. Softw. Eng. Methodol.* 21.1 (2011), 4:1–4:25. URL: https://doi.org/10.1145/2063239.2063243 (cit. on pp. 47–50).

[234] R. Sindhgatta. ‘Using an Information Retrieval System to Retrieve Source Code Samples’. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. Shanghai, China: Association for Computing Machinery, 2006, pp. 905–908. URL: https://doi.org/10.1145/1134285.1134448 (cit. on p. 25).

[235] J. Singer, T. C. Lethbridge, N. G. Vinson, N. Anquetil. ‘An examination of software engineering work practices’. In: *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada*. Ed. by J. H. Johnson. IBM, 1997, p. 21. URL: https://dl.acm.org/citation.cfm?id=782031 (cit. on pp. 47, 48).

[236] R. Sirres, T. F. Bissyande, D. Kim, D. Lo, J. Klein, K. Kim, Y. L. Traon. ‘Augmenting and structuring user queries to support efficient free-form code search’. In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by M. Chaudron, I. Crnkovic, M. Chechik, M. Harman. ACM, 2018, p. 945. URL: https://doi.org/10.1145/3180155.3182513 (cit. on pp. 20, 28, 30, 33, 37, 39, 44).

[237] B. Sisman, A. C. Kak. ‘Assisting code search with automatic Query Reformulation for bug localization’. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013, pp. 309–318 (cit. on pp. 28–30, 33, 39).

[238] A. Sivaraman, T. Zhang, G. V. den Broeck, M. Kim. ‘Active inductive logic programming for code search’. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by J. M. Atlee, T. Bultan, J. Whittle. IEEE, 2019, pp. 292–303. URL: https://doi.org/10.1109/ICSE.2019.00044 (cit. on pp. 25, 33, 34, 44).

[239] R. Sousa, G. Soares, R. Gheyi, T. Barik, L. D’Antoni. ‘Learning Quick Fixes from Code Repositories’. In: *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. SBES ’21. Joinville, Brazil: Association for Computing Machinery, 2021, pp. 74–83. URL: https://doi.org/10.1145/3474624.3474650 (cit. on p. 153).

[240] C. Staicu, M. Pradel. 'Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers'. In: *USENIX Security Symposium*. 2018, pp. 361–376 (cit. on p. 155).

[241] K.-J. Stol, B. Fitzgerald. 'The ABC of Software Engineering Research'. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (Sept. 2018). URL: https://doi.org/10.1145/3241743 (cit. on p. 5).

[242] K. T. Stolee, S. Elbaum, D. Dobos. 'Solving the Search for Source Code'. In: *ACM Trans. Softw. Eng. Methodol.* 23.3 (June 2014). URL: https://doi.org/10.1145/2581377 (cit. on pp. 26, 44).

[243] K. T. Stolee, S. G. Elbaum. 'Toward semantic search via SMT solver'. In: *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. Ed. by W. Tracz, M. P. Robillard, T. Bultan. ACM, 2012, p. 25. URL: https://doi.org/10.1145/2393596.2393625 (cit. on p. 26).

[244] K. T. Stolee, S. G. Elbaum, M. B. Dwyer. 'Code search with input/output queries: Generalizing, ranking, and assessment'. In: *J. Syst. Softw.* 116 (2016), pp. 35–48. URL: https://doi.org/10.1016/j.jss.2015.04.081 (cit. on pp. 26, 33, 35, 43, 44).

[245] C. Sun, Y. Li, Q. Zhang, T. Gu, Z. Su. 'Perses: syntax-guided program reduction'. In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by M. Chaudron, I. Crnkovic, M. Chechik, M. Harman. ACM, 2018, pp. 361–371. URL: https://doi.org/10.1145/3180155.3180236 (cit. on p. 139).

[246] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, Q. Zhang. 'Code Search Based on Context-Aware Code Translation'. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 388–400. URL: https://doi.org/10.1145/3510003.3510140 (cit. on pp. 20, 33, 35, 37, 40, 44, 45).

[247] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, L. Zhang. 'TreeGen: A tree-based transformer architecture for code generation'. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 05. 2020, pp. 8984–8991 (cit. on p. 157).

[248] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, M. M. Mia. 'Towards a Big Data Curated Benchmark of Inter-project Code Clones'. In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 476–480. URL: https://doi.org/10.1109/ICSME.2014.77 (cit. on p. 53).

[249] W. Takuya, H. Masuhara. 'A Spontaneous Code Recommendation Tool Based on Associative Search'. In: *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. SUITE '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 17–20. URL: https://doi.org/10.1145/1985429.1985434 (cit. on pp. 22, 23, 33, 36, 39, 45, 46, 53).

[250] S. H. Tan, J. Yi, Yulis, S. Mechtaev, A. Roychoudhury. 'Codeflaws: a programming competition benchmark for evaluating automated program repair tools'. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. Ed. by S. Uchitel, A. Orso, M. P. Robillard. IEEE Computer Society, 2017, pp. 180–182. URL: https://doi.org/10.1109/ICSE-C.2017.76 (cit. on p. 56).

[251] R. Tate, M. Stepp, Z. Tatlock, S. Lerner. 'Equality saturation: a new approach to optimization'. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Z. Shao, B. C. Pierce. ACM, 2009, pp. 264–276. URL: https://doi.org/10.1145/1480881.1480915 (cit. on p. 26).

[252] K. F. Tómasdóttir, M. Aniche, A. van Deursen. 'Why and how JavaScript developers use linters'. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 578–589 (cit. on p. 92).

[253] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, D. Poshyvanyk. 'On learning meaningful code changes via neural machine translation'. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, pp. 25–36. URL: https://dl.acm.org/citation.cfm?id=3339509 (cit. on pp. 153, 157).

[254] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, R. Singh. 'Neural Program Repair by Jointly Learning to Localize and Repair'. In: *ICLR*. 2019 (cit. on p. 157).

[255] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin. 'Attention is all you need'. In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 4).

[256] V. Vinayakarao, A. Sarma, R. Purandare, S. Jain, S. Jain. 'ANNE: Improving Source Code Search using Entity Retrieval Approach'. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*. Ed. by M. de Rijke, M. Shokouhi, A. Tomkins, M. Zhang. ACM, 2017, pp. 211–220. URL: https://doi.org/10.1145/3018661.3018691 (cit. on pp. 20, 21, 44).

[257] M. M. Vitousek, A. M. Kent, J. G. Siek, J. Baker. 'Design and evaluation of gradual typing for python'. In: *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by A. P. Black, L. Tratt. ACM, 2014, pp. 45–56. URL: https://doi.org/10.1145/2661088.2661101 (cit. on pp. 90, 156).

[258] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, H. Jin. 'What Do They Capture? A Structural Analysis of Pre-Trained Language Models for Source Code'. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2377–2388. URL: https://doi.org/10.1145/3510003.3510050 (cit. on p. 158).

[259] S. Wang, D. Lo, L. Jiang. 'Code Search via Topic-Enriched Dependence Graph Matching'. In: *2011 18th Working Conference on Reverse Engineering*. 2011, pp. 119–123 (cit. on pp. 25, 28, 31).

[260] S. Wang, D. Lo, L. Jiang. 'Active Code Search: Incorporating User Feedback to Improve Code Search Relevance'. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 677–682. URL: https://doi.org/10.1145/2642937.2642947 (cit. on p. 20).

[261] W. Wang, Y. Zhang, Z. Zeng, G. Xu. 'TranS3: A Transformer-based Framework for Unifying Code Summarization and Code Search'. In: *CoRR* abs/2003.03238 (2020). arXiv: 2003.03238. URL: https://arxiv.org/abs/2003.03238 (cit. on p. 20).

[262] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, J. X. Yu. 'Matching dependence-related queries in the system dependence graph'. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 457–466 (cit. on p. 25).

[263] Y. Wang, F. Gao, L. Wang. 'Demystifying Code Summarization Models'. In: *CoRR* (2021). URL: https://arxiv.org/abs/2102.04625 (cit. on p. 158).

[264] P. Weißgerber, S. Diehl. 'Identifying Refactorings from Source-Code Changes'. In: *International Conference on Automated Software Engineering (ASE)*. IEEE, 2006, pp. 231–240 (cit. on p. 152).

[265] M. Wen, R. Wu, S. Cheung. 'Locus: locating bugs from software changes'. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 2016, pp. 262–273. URL: https://doi.org/10.1145/2970276.2970359 (cit. on p. 152).

[266] D. Wightman, Z. Ye, J. Brandt, R. Vertegaal. 'SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization'. In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST '12. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2012, pp. 219–228. URL: https://doi.org/10.1145/2380116.2380145 (cit. on pp. 23, 27, 44, 46).

[267] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa. 'A survey on software fault localization'. In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740 (cit. on p. 152).

[268] H. Wu, Y. Yang. 'Code Search Based on Alteration Intent'. In: *IEEE Access* 7 (2019), pp. 56796–56802 (cit. on pp. 20, 28, 30, 33, 39, 44).

[269] R. Wu, M. Wen, S. Cheung, H. Zhang. 'ChangeLocator: locate crash-inducing changes based on crash reports'. In: *Empirical Software Engineering* 23.5 (2018), pp. 2866–2900. URL: https://doi.org/10.1007/s10664-017-9567-4 (cit. on p. 152).

[270] X. Wu, C. Zhu, Y. Li. 'DIFFBASE: A Differential Factbase for Effective Software Evolution Management'. In: *ESEC/FSE*. 2021 (cit. on p. 151).

[271] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, J. Dean. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. URL: https://arxiv.org/abs/1609.08144 (cit. on p. 37).

[272] C. S. Xia, Y. Wei, L. Zhang. 'Automated program repair in the era of large pre-trained language models'. In: *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*. 2023 (cit. on pp. 146, 157).

[273] C. S. Xia, L. Zhang. 'Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT'. In: *arXiv preprint arXiv:2304.00385* (2023) (cit. on pp. 146, 157).

[274] C. S. Xia, L. Zhang. 'Less training, more repairing please: revisiting automated program repair via zero-shot learning'. In: *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*. Ed. by A. Roychoudhury, C. Cadar, M. Kim. ACM, 2022, pp. 959–971. URL: https://doi.org/10.1145/3540250.3549101 (cit. on p. 157).

[275] X. Xia, X. He, Y. Yan, L. Xu, B. Xu. 'An Empirical Study of Dynamic Types for Python Projects'. In: *International Conference on Software Analysis, Testing, and Evolution*. Springer. 2018, pp. 85–100 (cit. on p. 92).

[276] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, Z. Xu. 'Two-Stage Attention-Based Model for Code Search with Textual and Structural Features'. In: *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 2021, pp. 342–353. URL: `https://doi.org/10.1109/SANER50967.2021.00039` (cit. on pp. 20, 34, 41).

[277] Z. Xu, P. Liu, X. Zhang, B. Xu. 'Python predictive analysis for bug detection'. In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. Ed. by T. Zimmermann, J. Cleland-Huang, Z. Su. ACM, 2016, pp. 121–132. URL: `https://doi.org/10.1145/2950290.2950357` (cit. on p. 156).

[278] Z. Xu, X. Zhang, L. Chen, K. Pei, B. Xu. 'Python probabilistic type inference with natural language support'. In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 2016, pp. 607–618. URL: `https://doi.org/10.1145/2950290.2950343` (cit. on pp. 90, 120, 156).

[279] F. Yang, H. Mei, K. Li. 'Software Reuse Software Component Technology'. In: *Acta Electronica Sinica* 27 (1999), pp. 68–75 (cit. on p. 14).

[280] Y. Yang, X. Xia, D. Lo, J. Grundy. 'A survey on deep learning for software engineering'. In: *ACM Computing Surveys (CSUR)* 54.10s (2022), pp. 1–73 (cit. on p. 4).

[281] M. Yasunaga, P. Liang. 'Graph-based, Self-Supervised Program Repair from Diagnostic Feedback'. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 10799–10808. URL: `http://proceedings.mlr.press/v119/yasunaga20a.html` (cit. on p. 157).

[282] F. Ye, S. Zhou, A. Venkat, R. Marcus, P. Petersen, J. J. Tithi, T. Mattson, T. Kraska, P. Dubey, V. Sarkar, J. Gottschlich. 'Context-Aware Parse Trees'. In: *arXiv preprint arXiv:2003.11118* (2020) (cit. on pp. 20, 44, 46).

[283] H. Ye, M. Martinez, M. Monperrus. 'Neural Program Repair with Execution-based Backpropagation'. In: *ICSE*. 2022 (cit. on p. 157).

[284] A. Zeller. 'Isolating cause-effect chains from computer programs'. In: *ACM SIGSOFT Software Engineering Notes* 27.6 (2002), pp. 1–10 (cit. on pp. 122, 132, 152).

[285] J. M. Zhang, F. Li, D. Hao, M. Wang, H. Tang, L. Zhang, M. Harman. 'A Study of Bug Resolution Characteristics in Popular Programming Languages'. In: *IEEE Trans. Software Eng.* 47.12 (2021), pp. 2684–2697. URL: https://doi.org/10.1109/TSE.2019.2961897 (cit. on p. 155).

[286] S. Zhou, H. Zhong, B. Shen. 'SLAMPA: Recommending Code Snippets with Statistical Language Model'. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 79–88 (cit. on pp. 22, 41, 44).

[287] S. Zhou, B. Shen, H. Zhong. 'Lancer: Your Code Tell Me What You Need'. In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 1202–1205. URL: https://doi.org/10.1109/ASE.2019.00137 (cit. on pp. 22, 23, 33, 34, 41, 46).

[288] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, L. Zhang. 'A syntax-guided edit decoder for neural program repair'. In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. Ed. by D. Spinellis, G. Gousios, M. Chechik, M. D. Penta. ACM, 2021, pp. 341–353. URL: https://doi.org/10.1145/3468264.3468544 (cit. on p. 157).

Appendix A
Curriculum Vitae

Luca Di Grazia

www.lucadigrazia.com
0000-0002-5306-8645

Major Achievements and Selected Awards

- **Until now:** Innovative research in Software Engineering and Generative AI at prestigious universities (Top-20 in CSRankings for Software Engineering) and a tech giant (Market Cap > $100B) across multiple countries.
- **2023:** Won *GenAI Uber's competition* with my internship project on using generative AI to boost developer productivity for fixing bugs, beating 103 teams, and presenting as winners to the Uber's CEO and ELT.
- **2022:** *ACM SIGSOFT Distinguished Paper Award* at ESEC/FSE 2022.
- **2022:** Second winner at *ACM Student Research Competition* at ICSE 2022 for my project: *Efficiently and Precisely Searching for Code Changes with DiffSearch* ($300).
- **2020:** Gnome Challenge 2020 winner (1st phase) to *Reach a new generation of open-source coders* ($1,000).
- **2016-2018:** Awarded national scholarship to study computer engineering at Polytechnic of Turin (€3,000/year).
- Reviewer for *IEEE TSE*, *ACM TOSEM*, Hiring Evaluator for the Max Planck Research School (IMPRS) for Intelligent Systems (IS) and the European Laboratory for Learning and Intelligent Systems Systems (ELLIS).

Selected Positions and Experience

From 02/2024 — **University of Lugano (USI)**, *Switzerland*
Role: Postdoctoral Researcher. Supervisor: Prof. Dr. Mauro Pezzè.

09/2019 – 02/2024 — **University of Stuttgart**, *Germany*
Role: Research and Teaching Assistant. Supervisor: Prof. Dr. Michael Pradel.

05/2023 – 08/2023 — **Uber**, *Amsterdam*, Netherlands
Role: Research intern (PhD) in Generative AI and AI Prompt Engineering. Supervisor: Dr. Raj Barik.

Education

09/2019 – 02/2024 — **University of Stuttgart**, *Germany*
Degree: Ph.D. in Computer Science, advisor Prof. Dr. Michael Pradel.

09/2013 – 07/2019 — **Polytechnic of Turin**, *Italy*
Degrees: Bachelor's and Master's Degree in Computer Engineering. Specialization in Embedded Systems.

February 6, 2024. *References and certificates on request.*

[illegible]
Luca Di Grazia

Appendix B

Erklärung

ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

Stuttgart, Deutschland, November 2023

Luca Di Grazia

[illegible]