DiffSearch: A Scalable and Precise Search Engine for Code Changes

Luca Di Grazia, Paul Bredl, Michael Pradel

Abstract—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 paper 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.

Index Terms—Software Engineering, Program Analysis, Software Maintenance.

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[1] 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 [1] or a program repair tool [2], [3], [4] 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 [5]. Finally, researchers systematically study when and how developers perform specific kinds of changes to increase our understanding of development practices [6], [7], [8], [9].

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 likely to result in irrelevant code changes. Another existing technique is GitHub Search,[2] 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 [10], [11], [12], [13]. 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 paper 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.

All authors are with the Department of Computer Science, University of Stuttgart, Germany. Email: luca.di-grazia@iste.uni-stuttgart.de, paulbredl@gmx.de, michael@binaervarianz.de

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

2. https://github.com/search

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* 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.[3]

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 [14], [15], test cases [16], or partial code snippets [17], [18]. 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 [10], [19], [20], [21], [22] and infers recurring code change patterns [8], [23]. 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 [24], [25], [26], [27], [28], 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 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 paper 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

2 Example and Overview

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 [29].

```
    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. 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.

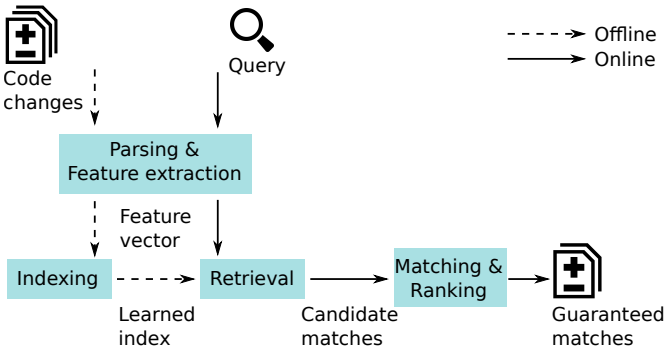


Fig. 1: Overview of the approach.

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 [30], often containing multiple entangled logical changes [11], [31], [32], [33]. 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 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 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:

Definition 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.

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 Figure 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 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 [34] 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 Approach

This section presents the approach in detail. Before going through the four components introduced in Section 2.3, we define the query language to specify what kind of code changes to search for.

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. [35]. 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 [35].

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 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.

Figure 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

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)

Fig. 2: Simplified grammar of queries. Non-terminals are in *italics*.

TABLE 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>); → runNow(EXPR<0>);`
	`now(EXPR<0>);`

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 1 gives a few examples of code changes and a corresponding query that matches the code change. The first two 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.

3.2 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 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 2). For example, Figure 3 shows the parse trees of a change and a query. The change on the left corresponds to Code change 2 from Section 2, which swaps `x` and `y` of a call to `isValidPoint`. Note that code edits that do not cause any change of the parse 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 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 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 [10], [13]. 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 Figure 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

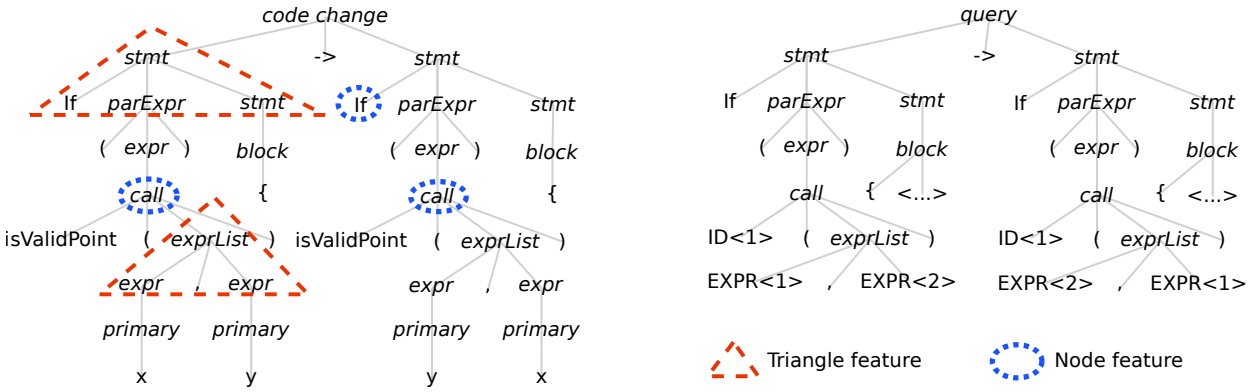


Fig. 3: Parse tree representations of Code change 2 (left) and the query from Section 2 (right). Only some of all considered features are highlighted for illustration.

Algorithm 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: $\quad h \leftarrow hash(f)$
4: $\quad v[h \bmod l_{target}] \leftarrow 1$
5: **return** v

default, i.e., a triangle contains a node and its immediate child nodes. For the example in Figure 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, 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 5.5 evaluates different sizes for the feature vector length.

Algorithm 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.

3.4 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 [34]. 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), 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 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.

3.5 Matching of Candidate Search Results

Given the k candidate code changes retrieved for a given query as described in Section 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 4 (Match). Given a code change $c \to c'$ and a query $q \to q'$, 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 again. They match because the tree on the right can be expanded into the tree on the left. 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 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`.

Algorithm 2 Check if a code change matches a query.

Input: Code change $c \to c'$ and query $q \to q'$
Output: True if they match, False otherwise.

```
1:  t_c, t_c' ← parse(c → c')
2:  t_q, t_q' ← parse(q → q')
3:  N_toMatch ← (allNodes(q) ∪ allNodes(q')) \ wildcards
4:  W ← candidateMappings(t_c, t_c', t_q, t_q')
5:  while W is not empty do
6:      M ← Take a mapping from W
7:      n_q ← nextUnmatchedNode(M, t_q, t_q')
8:      n_pq ← Parent of n_q
9:      n_pc ← 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' ← Copy of M with n_q ↦ c
13:             if keys(M') ∩ N_toMatch = ∅
14:               and isValid(M, t_c, t_c', t_q, t_q') then
15:                 return true
16:         else
17:             Add M' to W
```

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 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 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 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 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 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 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`:[4]

```
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 2, similar to Coccinelle [36].

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,[5] using the grammar of the target programming language, modified to support queries and to allow for syntactically incomplete code fragments (Section 3.1). The indexing and retrieval components build on the FAISS library [34], 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.[6]

5 Evaluation

Our evaluation focuses on six research questions:

- RQ1: What is the recall of DiffSearch? (Section 5.1)
- RQ2: How efficient and scalable is DiffSearch? (Section 5.2)
- RQ3: Does DiffSearch enable users to find relevant code changes more effectively than a regular expression-based search through raw diffs? (Section 5.3)
- RQ4: Is DiffSearch useful for finding examples of recurring bug fix patterns? (Section 5.4)
- RQ5: How do parameters of the approach influence the results? (Section 5.5)
- RQ6: How do queries and search results compare in terms of their size and absolute number? (Section 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 [37]. The experiments are performed on a server with 48 Intel Xeon CPU cores clocked at 2.2GHz, 250GB of RAM, running Ubuntu 18.04.

5.1 RQ1: Recall

While the precision of DiffSearch's results is guaranteed by design (Section 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

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

5. https://www.antlr.org/

6. http://diffsearch.software-lab.org

TABLE 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%**

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 2 precisely computes whether a code change matches a query. By design of DiffSearch (Section 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 2.

Table 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, 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 5.5 that the recall can be increased even further by retrieving more candidate matches, at the expense of a slightly increased response time.

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 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 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 [34] 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 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 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.

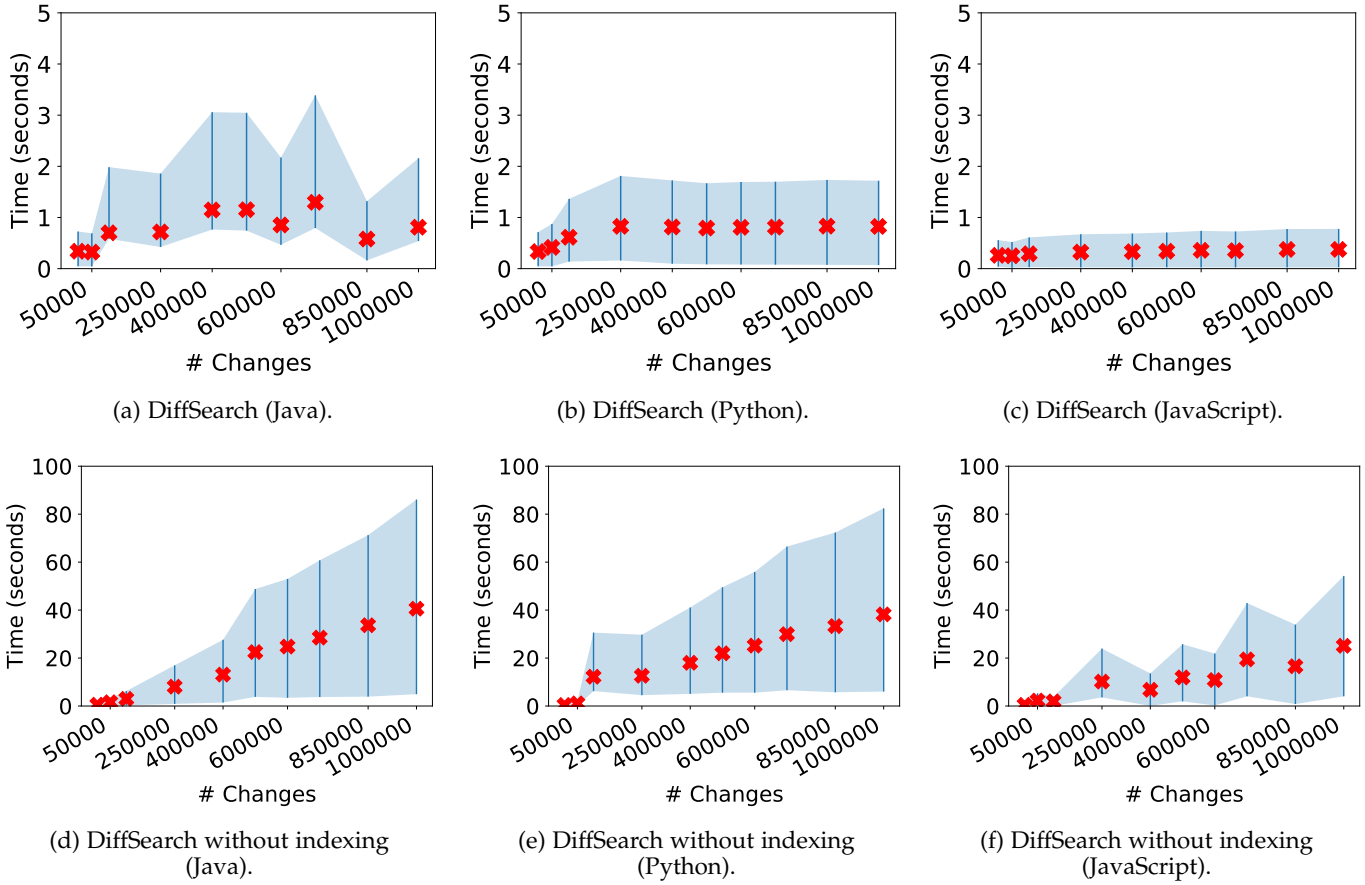
Even though scalability is most relevant for the online part of DiffSearch, we also measure how long the offline part takes. In total, analyzing a million code changes to extract feature vectors and indexing these vectors takes up to five hours. As this is a one-time effort that does not influence the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

5.3 RQ3: User Study

5.3.1 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 [38], 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) 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



(a) DiffSearch (Java). (b) DiffSearch (Python). (c) DiffSearch (JavaScript).

(d) DiffSearch without indexing (Java). (e) DiffSearch without indexing (Python). (f) DiffSearch without indexing (JavaScript).

Fig. 4: Response time across differently sized datasets (average and 95% confidence interval). Top: Full DiffSearch. Bottom: DiffSearch without indexing.

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 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[7] 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 this paper. 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.[8] 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

7. https://github.com/luca-digrazia/DatasetCommitsDiffSearch

8. The web interface is available to reviewers, see end of Section 1.

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.

5.3.2 Quantitative Results

Table 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.

5.3.3 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.

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 [1], automated program repair tools [5], or for training a learning-based bug detection tool [39]. We build on a set of 16 patterns defined by prior work [37], of which we use twelve (Table 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 [37] 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 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

TABLE 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

TABLE 4: Effectiveness of DiffSearch in finding instances of bug fix patterns [37].

	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

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.

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), and the number k of candidate matches retrieved via the feature vectors (Section 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 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.

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.

TABLE 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

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.

6 Limitations and Future Work

Our approach has some limitations that will be interesting to address in future work. First, in Section 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 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 [40] and Commit2Vec [41].

7 Related Work

Code Search. Code search engines allow users to find code snippets based on method signatures [16], existing code examples [17], [18], [42], or natural language queries [15], [43], [44]. Sourcerer provides an infrastructure that combines several of the above ideas [14]. Early work by Paul et al. [45] proposes a mechanism similar to the placeholders in our query language. The most important difference between these approaches and DiffSearch is that we search for changes of code, not for code snippets within a single snapshot of code. Another difference is that DiffSearch guarantees that all search results match the given query, whereas the existing techniques, with the exception of [42], are aimed at similarity only.

Prequel has a goal similar to DiffSearch, and matches C, and partially also C++ and Java, patches against user-provided rules that the code before and after a patch must comply with [13]. The approaches differ in four aspects. First, Prequel can describe all the parts of a commit, including the code surrounding and relationship between multiple hunks. Instead, DiffSearch focus on single hunks. Second, Prequel's rules are based on the semantic patch language of Coccinelle [36] and may include executable code, e.g., queries are Turing-complete. In contrast, our queries are purely declarative and build on the underlying programming language. Third, Prequel pre-filters commits based on a regular expression or indexing, followed by a linear search through all remaining commits. As a result, answering a query may take minutes or, if the pre-filtering is not effective, even longer [13]. In contrast, DiffSearch avoids a linear search via feature-based retrieval, and hence, responds to queries across hundreds of thousands of code changes within seconds. The indexing module could be used also in Prequel in principle. Finally, Prequel is evaluated on a different problem than DiffSearch: obtaining examples to motivate device driver porting from 300,000 commits of the Linux kernel.

Several ideas to improve the user's interaction with a code search engine have been proposed, such as refining search results based on user feedback about the quality of results [46], [47]. Other work resolves vocabulary mismatches between queries and code [48]. Finally, code2vec [49] represents a code snippet with a single-size feature vector, but with respect to DiffSearch they use ASTs and neural networks. The main difference with our work is that code2vec converts snippets of code in vectors, instead DiffSearch converts code changes in vectors. Future work could adopt similar ideas to searching for code changes.

Code Changes as Edit Scripts. To reason about code changes, several techniques derive edit scripts on ASTs [10],

[19], [22], [50], providing an abstract description of the change that can then be applied elsewhere [51]. Lase generalizes from multiple code changes into a single edit script [52]. Future work 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 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.

Mining Code Changes. Work on mining code repositories and learning from code changes shows development histories to be a rich source of implicitly stored knowledge. For example, existing approaches leverage version histories to extract repetitive code changes [6], [8], [53], predict code changes [54], predict bugs [55], [56], or to learn about API usages [57], [58]. 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.

Learning from Code Changes. 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 [5], [20], [59]. Another line of work ranks potential program repairs based on their similarity to common code change patterns [60]. DiffSearch could help gather datasets of changes for these approaches to learn from, e.g., based on queries for bug fixing patterns.

Representation Learning on Commits. The feature extractor of DiffSearch relates to techniques for learning vector representations of commits, such as CC2Vec [40] and Commit2Vec [41]. 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 [40] or whether a commit is labeled as security-critical [41]. 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.

Other Analyses of Code Changes. There are various other analyses of code changes, of which we discuss only a subset here. Hashimoto et al. propose a technique for reducing a diff to the essence of a bug [61]. Nielsen et al. [62] use JavaScript code change templates to fix code broken due to library evolution. Another approach automatically documents code changes with a natural language description [63]. SCC [64] and DeepJIT [65] are predictive models that estimate how likely a code change is to introduce a bug. A related problem is to find the bug-inducing code change for a given bug report [66], [67]. DiffBase [68] encodes facts about different versions of a program to facilitate multi-version program analyses. CodeShovel [69] tracks a method from its creation to its current state throughout a version history. All these approaches relate to our work by also reasoning about code changes, but they aim for different goals than DiffSearch.

Clone Detection. DiffSearch relates to code clone detectors [24], [25], [26], [27], [28], 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 [28]. 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 [26] computes characteristic vectors of parse trees and SourcererCC [28] 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. [70] 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. [71] 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.

8 Conclusion

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.

ACKNOWLEDGMENT

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects.

REFERENCES

[1] A. Habib and M. Pradel, "How many of all bugs do we find? A study of static bug detectors," in *ASE*, 2018.

[2] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

[3] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and 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*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE Computer Society, 2017, pp. 180–182. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.76

[4] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE Transactions on Software Engineering*, 2020.

[5] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," in *OOPSLA*, 2019, pp. 159:1–159:27.

[6] S. Negara, M. Codoban, D. Dig, and 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.

[7] I. Rak-amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby, "Python 3 types in the wild: A tale of two type systems," in *DLS*, 2020.

[8] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and 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. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00089

[9] A. Eghbali and 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. [Online]. Available: https://ieeexplore.ieee.org/document/9286132

[10] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[11] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 351–360.

[12] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[13] J. Lawall, D. Palinski, L. Gnirke, and 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.

[14] S. K. Bajracharya, T. C. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. V. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, P. L. Tarr and W. R. Cook, Eds. ACM, 2006, pp. 681–682. [Online]. Available: https://doi.org/10.1145/1176617.1176671

[15] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 933–944. [Online]. Available: https://doi.org/10.1145/3180155.3180167

[16] 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. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070525

[17] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, p. 152, 2019.

[18] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 946–957.

[19] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and 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. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[20] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and 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. [Online]. Available: https://doi.org/10.1109/ICSE.2017.44

[21] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, and A. Tiwari, "Feedback-driven semi-supervised synthesis of program transformations," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 219:1–219:30, 2020. [Online]. Available: https://doi.org/10.1145/3428287

[22] S. Erdweg, T. Szabó, and A. Pacak, "Concise, type-safe, and efficient structural diffing," in *PLDI*, 2021.

[23] R. Paletov, P. Tsankov, V. Raychev, and 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*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 450–464. [Online]. Available: https://doi.org/10.1145/3192366.3192403

[24] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[26] L. Jiang, G. Misherghi, Z. Su, and 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.

[27] C. K. Roy and 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.

[28] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and 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.

[29] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.

[30] A. Alali, H. H. Kagdi, and 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*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. IEEE Computer Society, 2008, pp. 182–191. [Online]. Available: https://doi.org/10.1109/ICPC.2008.24

[31] M. Barnett, C. Bird, J. Brunet, and 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, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 134–144. [Online]. Available: https://doi.org/10.1109/ICSE.2015.35

[32] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empir. Softw. Eng.*, vol. 21, no. 2, pp. 303–336, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9376-6

[33] P. Pârtachi, S. K. Dash, M. Allamanis, and 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*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 63–74. [Online]. Available: https://doi.org/10.1145/3368089.3409693

[34] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[35] L. Di Grazia and M. Pradel, "Code search: A survey of techniques for finding code," *ACM Comput. Surv.*, sep 2022. [Online]. Available: https://doi.org/10.1145/3565971

[36] J. Lawall and 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*, H. S. Gunawi and B. Reed, Eds. USENIX Association, 2018, pp. 601–614. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/lawall

[37] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 573–577. [Online]. Available: https://doi.org/10.1145/3379597.3387491

[38] C. Sadowski, K. T. Stolee, and 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.

[39] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *PACMPL*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018. [Online]. Available: https://doi.org/10.1145/3276517

[40] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[41] R. Cabrera Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2vec: Learning distributed representations of code changes," *SN Computer Science*, vol. 2, no. 3, pp. 1–16, 2021.

[42] V. Premtoon, J. Koppel, and A. Solar-Lezama, "Semantic code search via equational reasoning," in *PLDI*, 2020.

[43] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and 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.

[44] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[45] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, 1994.

[46] L. Martie, A. v. d. Hoek, and 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*, 2017, pp. 774–785.

[47] A. Sivaraman, T. Zhang, G. Van den Broeck, and M. Kim, "Active inductive logic programming for code search," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 292–303.

[48] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. Le Traon, "Augmenting and structuring user queries to support efficient free-form code search," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2622–2654, 2018.

[49] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019. [Online]. Available: https://doi.org/10.1145/3290353

[50] M. Hashimoto and 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. [Online]. Available: https://doi.org/10.1109/WCRE.2008.44

[51] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example." in *PLDI*, 2011, pp. 329–342.

[52] ——, "Lase: locating and applying systematic edits by learning from examples." in *ICSE*, 2013, pp. 502–511.

[53] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and 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.

[54] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and 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. [Online]. Available: https://dl.acm.org/citation.cfm?id=3339509

[55] V. B. Livshits and 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.

[56] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[57] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and 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*, 2016, pp. 511–522.

[58] R. Paletov, P. Tsankov, V. Raychev, and 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*, 2018, pp. 450–464.

[59] R. Sousa, G. Soares, R. Gheyi, T. Barik, and L. D'Antoni, "Learning quick fixes from code repositories," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, ser. SBES '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 74–83. [Online]. Available: https://doi.org/10.1145/3474624.3474650

[60] X. D. Le, D. Lo, and 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. [Online]. Available: https://doi.org/10.1109/SANER.2016.76

[61] M. Hashimoto, A. Mori, and 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. [Online]. Available: https://doi.org/10.1145/3236024.3236047

[62] B. B. Nielsen, M. T. Torp, and 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.

[63] R. P. L. Buse and W. Weimer, "Automatically documenting program changes," in *Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 33–42.

[64] E. Giger, M. Pinzger, and 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.

[65] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and 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. [Online]. Available: https://doi.org/10.1109/MSR.2019.00016

[66] M. Wen, R. Wu, and 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. [Online]. Available: https://doi.org/10.1145/2970276.2970359

[67] R. Wu, M. Wen, S. Cheung, and H. Zhang, "Changelocator: locate crash-inducing changes based on crash reports," *Empirical*

Software Engineering, vol. 23, no. 5, pp. 2866–2900, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9567-4

[68] X. Wu, C. Zhu, and Y. Li, "Diffbase: A differential factbase for effective software evolution management," in *ESEC/FSE*, 2021.

[69] F. Grund, S. A. Chowdhury, N. Bradley, B. Hall, and R. Holmes, "Codeshovel: Constructing method-level source code histories," in *ICSE*, 2021.

[70] K. Inoue, Y. Miyamoto, D. M. German, and T. Ishio, "Code clone matching: A practical and effective approach to find code snippets," *arXiv preprint arXiv:2003.05615*, 2020.

[71] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and 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.

Luca Di Grazia is a PhD student in the Department of Computer Science at the University of Stuttgart in Germany, advised by Michael Pradel. His main research interests are code evolution and maintenance, mining software repositories, and program analysis.

Paul Bredl is a master student in Software Engineering at the University of Stuttgart. During and after his bachelor studies he worked as a software developer, maintaining and extending enterprise systems.

Michael Pradel is a full professor at the University of Stuttgart. His research interests span software engineering, programming languages, security, and machine learning, with a focus on tools and techniques for building reliable, efficient, and secure software.