Compiler-directed Migrating API Callsite of Client Code

Hao Zhong
Shanghai Jiao Tong University, China
zhonghao@sjtu.edu.cn

Na Meng
Virginia Polytechnic Institute and State University, USA
nm8247@cs.vt.edu

ABSTRACT

API developers evolve software libraries to fix bugs, add new features, or refactor code, but the evolution can introduce API-breaking changes (*e.g.*, API renaming). To benefit from such evolution, the programmers of client projects have to repetitively upgrade the callsites of libraries, since API-breaking changes introduce many compilation errors. It is tedious and error-prone to resolve such errors, especially when programmers are often unfamiliar with the API usages of newer versions. To migrate client code, the prior approaches either mine API mappings or learn edit scripts, but both the research lines have inherent limitations. For example, mappings alone cannot handle complex cases, and there is no sufficient source (*e.g.*, migration commits) for learning edit scripts.

In this paper, we propose a new research direction. When a library is replaced with a newer version, each type of API-breaking change introduces a type of compilation error. For example, renaming the name of an API method causes undefined-method errors at its callsites. Based on this observation, we propose to resolve errors that are introduced by migration, according to their locations and types that are reported by compilers. In this way, a migration tool can incrementally migrate complex cases, even without any change examples. Towards this direction, we propose the first approach, called LibCatch. It defines 14 migration operators, and in a compiler-directed way, it exploits the combinations of migration operators to generate migration solutions, until its predefined criteria are satisfied. We conducted two evaluations. In the first evaluation, we use LibCatch to handle 123 migration tasks. LibCatch reduced migration-related compilation errors for 92.7% of tasks, and eliminated such errors for 32.4% of tasks. We inspect the tasks whose errors are eliminated, and find that 33.9% of them produce identical edits to manual migration edits. In the second evaluation, we use two tools and LibCatch to migrate 15 real client projects in the wild. LibCatch resolved all compilation errors of 7 projects, and reduced the compilation errors of 6 other projects to no more than two errors. As a comparison, the compared two tools reduced the compilation errors of only 1 project.

CCS CONCEPTS

• **Software and its engineering**; • **Software creation and management** → *Software evolution*; • **Software notations and tools** → Software maintenance tools;

KEYWORDS

Code migration, Compiler, and API library

ACM Reference Format:
Hao Zhong and Na Meng. 2024. Compiler-directed Migrating API Callsite of Client Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639084

1 INTRODUCTION

API developers constantly repair bugs and implement new features for libraries. To benefit from a new library, programmers shall update their code to call newer versions. For example, the cybersecurity & infrastructure security agency (CISA) warns that the notorious vulnerability, `CVE-2021-44228`, affects Apache `log4j 2.0-beta9` to `2.14.1` [4]. After Apache developers fix this vulnerability, they ask client programmers to update their libraries and call newer versions [3]. When updating libraries, programmers must resolve many compilation errors, since new libraries contain API-breaking changes [43, 67]. It is tedious, repetitive, and error-prone to resolve such compilation errors. Programmers typically are unfamiliar with the API usages of newer versions, but only about 20% of API-breaking changes are documented [20].

Researchers have proposed various approaches to assist the migration of client code, and these approaches can be roughly divided into two research lines. The first line of approach mines API mappings from clients, libraries, and other sources. For example, Dagenais and Robillard [21] infer API method mappings by comparing how API methods are called by two versions of libraries. If API methods are renamed, after the library is replaced, their callsites cause undefined method errors. Replacing old API methods with new ones can reduce undefined method errors, but mappings alone are insufficient to migrate many other compilation errors caused by API-breaking changes. The second line of approaches infers edit scripts from change examples, and applies them to new code locations. As a typical approach in this research line, Meng *et al.* [51] can handle more complicated edits. However, change examples are often unavailable or expensive to craft manually. To handle this problem, researchers extract migration commits as change examples [24, 78]. To support the migration from `v1` to `v2`, the library versions of migration commits must be exactly from `v1` to `v2`, since other versions of libraries typically provide different APIs. Even from large code repositories and with the support of advanced techniques [41, 59], it is unlikely to extract many exactly matched migration commits for learning. For example, in total, Xu *et al.* [78] extract 3,674 migration commits from 465 projects. Among these commits, Xu *et al.* [78] report that their most frequent Android version pair (`19-21`) has only 63 migration commits. Migrating real Android applications must handle more API-breaking changes. The limitations of the above two research lines are inherent.

Each type of API-breaking change causes its corresponding compilation errors. For example, renamed API methods cause undefined method errors, and can be resolved by replacing methods. Besides providing hints to migrate client code, compilers report which error locations require migration edits. Based on these observations, instead of following the prior directions, we redefine the problem of migrating client code and propose a new research direction. This paper makes the following contributions:

- **A new research direction that uses compilation errors to guide the migration process.** We reduce the migration process to an optimization problem [14], and our direction does not suffer from the limitations of the prior approaches. Besides the definition itself, the inputs of our directions are both different from the prior research lines.
- **The first approach, called LibCatch, that illustrates our new direction.** We propose an pragmatic, sensible approaches to migrate API callsites based on our predefined migration operators, and it can handle more than 94% of API-breaking changes that were reported by the prior studies [15, 16, 36]. To handle complex cases, we further propose a migration algorithm to guide the migration process. It applies our migration operators incrementally, according to the types and locations of compilation errors.
- **Positive results on our tasks.** Among our 123 migration tasks, LibCatch reduced the compilation errors of 114 tasks (95.1%), and removed all the compilation errors of 61 tasks (49.6%). Among the 61 tasks, our edits of 24 tasks are identical to those from programmers (39.3%). In 33 tasks, our edits resolve all compilation errors, but programmers forget to update them (54.1%).
- **Migrated real projects in the wild.** As the first attempt to migrate real projects, we use LibCatch and two prior approaches [21, 78] to migrate 15 real projects. LibCatch resolved all compilation errors in 7 projects, and reduced most errors in 6 projects. As a comparison, in total, as an API mapping approach, SemDiff [21] resolves only 2 errors from 1 project, and as a learning script approach, Meditor [78] resolves no compilation errors.

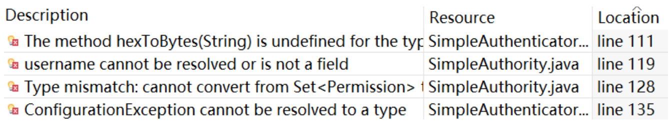
More details of the evaluations are listed on our website: https://github.com/drhaozhong/libcatch.

2 RELATED WORK

The prior approaches fall into two research lines:

Mining API mapping. This research line mines API mappings between two versions of a library. Between two versions, some APIs are unchanged. Wu *et al.* [74] extract mappings of changed methods based on how they call unchanged methods. Dagenais and Robillard [21] extend their approach with more advanced matching algorithms. Chen *et al.* [17] encode API calls and comments into vectors and mine mappings by their distances. Meng *et al.* [52] compare the revisions of libraries to mine API mappings. Kalra *et al.* [38] match execution traces of two libraries to detect their mappings. Xing and Stroulia [76] use the UMLDiff algorithm [77] to infer API mappings from recorded UML changes. Besides the mappings in the same language, researchers [56–58, 80] also mine API mappings across languages. Balaban *et al.* [13] proposed an approach to modify client code when mapping relations of libraries are already available. Liu *et al.* [46] recommend similar APIs based knowledge graphs. The prior studies [20, 44] show that replacements alone are insufficient to migrate many cases, but our approach can generate more complicated edits than replacements.

Learning edit scripts. This research line learns edit scripts from given change examples. Given an original file and its modified file, Andersen *et al.* [12] extract a set of term replacements. Meng *et al.* [50, 51] learn edit scripts that support more complicated edits than replacements. Rolim *et al.* [66] search for a transformation that is consistent with all given change samples. Long *et al.* [47] infer AST templates from patches. Nguyen *et al.* [59] mine graph change patterns from change examples. Given only a change example, Jiang *et al.* [37] and Haryono *et al.* [31] mine where and how to apply its transformation. Mesbah *et al.* [53] learn edit scripts from bug fixes and use learned scripts to repair compilation errors. Gao *et al.* [28] introduce clustering techniques to learn better scripts. Chow and Notkin [19] apply client-code changes if API changes and transformation rules are manually defined. Henkel and Diwan [34] capture and replay API refactoring actions to update the client code. Fazzini *et al.* [24] learned scripts to migrate Android client code, and Haryono *et al.* [32] migrate Python machine-learning APIs. Ketkar *et al.* mine type mappings from a code repository [40] and apply mined type mappings systematically [39]. Xu *et al.* [78] learn edit scripts from migration commits. Wasserman [71] proposes a tool called Refaster that refactors code based on given examples. Ossendrijver *et al.* [62] extend Refaster to migrate client code. Although edit scripts can handle complicated edits, it is challenging to extract sufficient change examples or migration commits for mining, but our approach does not suffer from this limitation.



Description	Resource	Location
The method hexToBytes(String) is undefined for the typ	SimpleAuthenticator...	line 111
username cannot be resolved or is not a field	SimpleAuthority.java	line 119
Type mismatch: cannot convert from Set<Permission> t	SimpleAuthority.java	line 128
ConfigurationException cannot be resolved to a type	SimpleAuthenticator...	line 135

(a) The reported compilation errors

```
-authenticated = MessageDigest.isEqual(..., FBUtilities.
        hexToBytes(props.getProperty(username)));
+authenticated = MessageDigest.isEqual(..., ByteBufferUtil.
        hexToBytes(props.getProperty(username)).array());
```

(b) The migration of `hexToBytes`

```
-if (reader.equals(user.username))
+if (reader.equals(user.getName()))
```

(c) The migration of `username`

Figure 1: Migration instances of our example

3 MOTIVATING EXAMPLE

The `examples` directory of `cassandra 1.0.0` provides an API example. We replace its `cassandra` library with `3.0.0`, and the replacement causes 20 compilation errors. These compilation errors provide valuable hints for migration. Xing and Stroulia [76] list such compilation errors and ask programmers to apply migration edits.

We notice that it is feasible to apply migration edits according to the types of compilation errors. Although applying an edit can introduce more compilation errors, we can reduce the migration process to an optimization problem if we define a suitable fitness

Table 1: API-breaking changes

Brito *et al.* [16]			Brito *et al.* [15]			Jezek *et al.* [36]		
Type	%	MA	Type	%	MA	Type	%	MA
DM	44%	MA1,MA2,MA4,MA7,MA14	MM	19%	MA1,MA2,MA4,MA7	IC	18%	MA1,MA2
RF	12%	MA1,MA2,MA3	DC	17%	MA1	IM	16%	MA1,MA2,MA4,MA7,MA10
RM	6%	MA1,MA2,MA4,MA7	CPL	15%	MA6	AMC	16%	MA13
CRT	6%	MA5	RM	14%	MA1,MA2,MA4,MA7	DM	15%	MA1,MA2,MA4,MA7,MA14
CFDV	6%	n/a	MC	14%	MA1,MA2	DC	14%	MA1
			AFM	10%	MA11	DF	13%	MA1,MA2,MA3
			DM	5%	MA1,MA2,MA4,MA7,MA14	IF	8%	MA1,MA2,MA3
			CRT	3%	MA5			
			CFDV	2%	n/a			
			AMC	2%	MA13			

DM: deleted method; MM: moved method; RM: renamed method; IM: incompatible method (*e.g.*, changed parameter types); DC: deleted class; MC: moved class; IC: incompatible class; DF: deleted field; RF: renamed field; IF: incompatible field; AMC: access modifier change; CRT: change in return type; CFDV: changes in field default value; CPL: changed parameter list; AFM: added final modifier.

Table 2: Our migration operators.

Category	Id	Migration action	Target compilation error
Missing API elements (ME)	MA1	Replacing undefined API elements with mappings	undefined types, methods, and variables
	MA2	Replacing undefined API elements with compatible ones	undefined methods and variables
	MA3	Replacing undefined fields with getters/setters	undefined fields
	MA4	Replacing undefined constructors with creators	undefined constructors
Incompatible API elements (IE)	MA5	Generating explicit conversions	class hierarchy changes (incompatible types)
	MA6	Reducing or swapping method parameters	incompatible methods
	MA7	Replacing static calls with instance calls and vice versa	undefined and incompatible methods
	MA8	Exploring declared fields and methods	incompatible actual parameters
More or fewer API calls (MF)	MA9	Generating method stubs	unimplemented methods
	MA10	Handling exceptions	unhandled exceptions
	MA11	Removing API calls	undefined methods and variables
Other issues (Other)	MA12	Resolving ambiguous types	ambiguous types
	MA13	Replacing invisible fields with getters and setters	invisible fields
	MA14	Removing `@Override` annotations	deleted methods in super types

function. The prior empirical studies [15, 16, 36] report the types of introduced compilation errors. To fulfil our vision, we design migration operators, and our operators cover all their reported compilation errors. For example, as the `hexToBytes` method is moved from the `FBUtilities` class to the `ByteBufferUtil` class, its callsites produce undefined method errors as shown in the first row of Figure 1a. For this compilation error, we define a migration operator that replaces undefined methods with similar methods of the new library. As this migration operator fits the compilation error, LibCatch replaces the call of the `hexToBytes` method with a new method declared by the `ByteBufferUtil` class. Although the mapping is correct, the replacement causes a type-mismatch error, since the return type of the `hexToBytes` method is modified from an array to `ByteBuffer`. Furthermore, we design a migration algorithm to combine simple edits into more complicated edits. For example, based on another migration operator that matches input and output types, LibCatch explores all the methods that are declared by `ByteBuffer`, and calls the `array` method as shown in Figure 1b. As another example, as the `AuthenticatedUser` class hides the `username` field, the accesses of this fields produce unresolved field errors as shown in the second row of Figure 1a. LibCatch finds that the `AuthenticatedUser` class declares a `getName()` method, and replaces the `username` field access with this method call as shown in Figure 1c. Section 4 introduces how LibCatch works, and Section 5 presents more migrated examples.

API mappings alone are insufficient to migrate this task. As shown in this example, replacing the old `hexToBytes` method declared by `FBUtilities` with its mapped method declared by `ByteBufferUtil` causes another compilation error, since its mapped method returns a different type. Learning edit scripts from migration commits can support challenging migrations [50, 51]. However, if a migration commit is useful, this commit must migrate the `cassandra` library exactly from `1.0.0` to `3.0.0`. As `cassandra` has many versions, it is unlikely to obtain sufficient useful migration commits. Section 6 presents more comparisons with the two research lines.

4 APPROACH

We use s to denote an initial solution whose library is replaced with a newer version. To resolve the compilation errors in s, we define a set of migration operators (Δ), and we use δ to denote a migration operator. If a solution has compilation errors, based on the types of compilation errors, we select the suitable migration operators, and apply them to the corresponding error locations. For a given initial solution (s), after applying δ on s, s is modified to a new solution, $s' \in S'$, where S' is the set of new solutions. When programmers migrate the initial solution, they must resolve all its compilation errors. As a result, for a candidate solution (s'), we define its fitness function value as the number of compilation errors $f(s')$. We thus reduce the migration process to an optimization problem that minimizes the following function:

$$\min_{s' \in S'} f(s') \qquad (1)$$

Our approach includes predefined migration operators (Section 4.1) and a guidance algorithm (Section 4.2).

4.1 Migration Operator

Table 1 lists the API-breaking changes reported by the prior studies [15, 16, 36] and our corresponding migration operators. For example, `DM` denotes that an API method is deleted. If it is a constructor, `MA5` resolves the problem; if it is a static method, `MA7` resolves it; and `MA1` and `MA2` handle the other cases. Among all the breaking changes, only the changes in field default values (CFDVs) are unhandled by our migration operators, and they account for 2% to 6% of the total changes. As they introduce no compilation errors, they are not API-breaking changes, and other researchers (*e.g.*, [15, 16, 36]) do not consider CFDVs as API-breaking changes either. CFDVs is a type of behavioral backward incompatibility as reported by Mostafa *et al.* [55].

Table 2 shows our migration operators and target compilation errors. For each category, we recreate its challenges when designing its migration operators. When we design our migration operators, we learn how the prior approaches in Section 2 resolve corresponding compilation errors. After that, we learn how to handle related compilation errors, and construct our migration operators. We implement our migration operators on Spoon [63], which is a library that allows the analysis and modifications of Java code. Although our initial solutions have compilation errors, it is still able to generate correct modifications, since replacing libraries does not introduce syntactical errors.

4.1.1 Missing API elements.

This category includes undefined types, undefined methods, undefined fields, invisible code elements, and incompatible methods. LibCatch implements the following migration operators to resolve these types of compilation errors:

MA1. Replacing API elements with mappings. For two API elements of the same type, LibCatch uses the Levenshtein edit distance of their full code names to calculate their distance. The Hungarian algorithm [42] is a classical algorithm to extract the best mappings between two sides of items. This algorithm has been used to compare the graphs of buggy and fixed files [81]. LibCatch uses the Hungarian algorithm to search for the mappings that can minimize the overall distance. The prior approaches mine API mappings from clients, documents, and traces. They will fail when such sources are unavailable. As a comparison, we can mine more mappings, since all APIs have names. For an undefined element, MA1 queries mined API mappings using the full name of the code element as the keyword. If a replacement is found, MA1 replaces the undefined element with the replacement. For the example in Section 3, LibCatch replaces the missing `IAuthority` interface with the `IAuthenticator` interface, since the mapping `IAuthority`⟶`IAuthenticator` is extracted. Like most approaches in the first research line of Section 2, LibCatch mines only one-to-one mappings, but it can combine simple edits to migrate complicated cases (see Section 4.2).

MA2. Replacing undefined API elements with compatible ones. If a method or a field is undefined, MA2 searches the code elements of the newer library, to locate its compatible matches. A compatible code element has a similar name, and introduces no more compilation errors, after it replaces the missing one. The similarity is defined as the reciprocal of the Levenshtein edit distance between code names. Based on our empirical results, we consider only the top ten similar items. Our other migration operators have other default values and settings, but we cannot evaluate all their impact due to space limit. We list all these issues in Section 8, and leave the tuning problem to future work.

MA3. Replacing undefined fields with getters or setters, and vice versa. If a field is missing, MA3 will replace it with its getters or setters. When this happens, a field name is typically similar to the names of its getter and setter. For example, the sample of Section 3 uses the `username` field of the `AuthenticatedUser` type. In `3.0.0`, this public field is changed to private. To handle the problem, LibCatch replaces the field with the `getName` method. Here, it first removes `get` or `set` from getters and setters. After that, it calculates the Levenshtein edit distances between the remaining method names and the field names, and selects the one with the least distance. Meanwhile, if a getter or a setter of a field is deleted, LibCatch will try to replace it with the field.

MA4. Replacing undefined constructors with creators. To implement a Factory design pattern, API developers can delete or hide the constructors of a class, and implement creators for the class. MA4 will replace such deleted and hidden constructors with their creators. For example, the samples of `cassandra 0.8.8` call the constructor of the `ColumnFamily` type, but the constructor is hidden in later versions. Instead, `ColumnFamily` implements a set of static methods to create the type. LibCatch replaces the constructor with these creators to resolve the compilation error. LibCatch determines that a method is a creator, if its name contains "create" and its return type is the declaring class.

4.1.2 Incompatible API elements.

This category includes conversion errors, and incompatible/undefined methods. LibCatch implements the following migration operators:

MA5. Generating explicit conversions. To resolve type conversion errors, MA5 adds cast expressions. For example, the types of following statement are mismatched:

```
authorized = Permission.ALL;
```

To resolve the problem, LibCatch adds an explicit cast:

```
authorized = (EnumSet<Permission>) Permission.ALL;
```

MA6. Reducing or swapping method parameters. When the signature of a method is changed, JDT can report incorrect overridden methods if the method is overridden, or incompatible methods if the method is directly called. Lamothe and Shang [44] also notice the problem, and introduce a case (*i.e*, the `queue(Bytebuffer, int)` method), in which the second actual parameter must be removed. Following their suggestions, LibCatch compares the new signature of its actual parameters of a callsite. If a parameter is deleted, LibCatch removes its corresponding actual parameter from the callsite. Alternatively, if the parameter order is changed, LibCatch reorders the parameters based on parameter types.

MA7. Replacing static calls with instance calls, and vice versa. If a static method is deleted, MA7 will replace it with instance methods, and vice versa. For example, the samples of `cassandra 0.8.0` call `ByteBufferUtil.bytes(key)` to obtain the bytes of `key`, but later versions delete the static method. The type of `key` is `Text`, and in the later versions, an instance method `getBytes()` is added to `Text` to obtain the bytes of `key`. To resolve the error, LibCatch replaces the static method call with `key.getBytes()`.

MA8. Exploring declared methods and fields of an incompatible type. If a method has an actual parameter whose type is `Type1` but the desirable one is `Type2`, MA8 explores all the fields and methods that are declared by `Type1`. If the type of a field is `Type2`, MA8 modifies the actual parameter to reference this field. Similarly, if the return type of a method is `Type2`, MA8 modifies the actual parameter to call this method.

4.1.3 More or fewer API calls. Programmers can add or delete API calls to resolve compilation errors during migration. LibCatch imitates the process to resolve these errors.

MA9. Generating method stubs. Suppose that a client-code class (`CC`) extends an API abstract class (`AC`) and implements an interface (`AI`), and in a follow-up version, a new abstract method is added to `AC`, or a new method is added to `AI`. `CC` must implement the newly added method in both cases to avoid compilation errors. If JDT reports that a type (`C`) does not implement a method (`m`), MA9 searches the super classes of `c` for the signature of `m`. With the found signature, LibCatch generates a method stub for `C`. For example, the example of `0.8.0` has a `SimpleAuthenticator` class, which implements the `IAuthenticator` interface. A later version adds more methods to the interface, but the class does not implement the added methods. As a result, the migrated example produces a compilation error: "The type `SimpleAuthenticator` must implement the inherited abstract method `IAuthenticator.requireAuthentication()`". To resolve the problem, LibCatch generates a method stub inside `SimpleAuthenticator`:

```
public boolean requireAuthentication() {
//todo: Please implement the method.
  return false;}
```

After the method stubs are generated, programmers have to implement its method body. As this migration operator indicates, programmers have to implement new code, when they migrate code to call newer APIs.

MA10. Handling exceptions. If the thrown exceptions of a method are modified, JDT can report exception-related errors. MA10 adds `throw` expressions or `try-catch` statements at error locations to remove such errors. For example, the `keyspace` method of `cassandra 1.2.0` throws two exceptions, but later versions throw three exceptions. As a result, when we migrate the examples of `cassandra 1.2.0` to later versions, JDT reports a compilation error that an exception is not handled. LibCatch produces a solution with an added `throw` expression and a solution with an added `try-catch` statement.

MA11. Removing API calls. Programmers can remove API calls, since their corresponding API elements are removed. For example, an example of `poi 3.16` calls the following method:

```
boldFont.setBoldweight(...COLOR_NORMAL.BOLDWEIGHT_BOLD);
```

As the later versions delete `BOLDWEIGHT_BOLD`, the above code produces a compilation error during migration. In the newer version of this example, programmers delete this code line to resolve the compilation error. Section 5.3.2 provides another similar example from `cassandra 1.2.0-beta3`. This migration operator imitates the above manual edits. To reduce the possibility of removing useful code lines, LibCatch removes at most one line each time. As removing useful code lines introduces compilation errors, applying this migration operator often produces solutions with more compilation errors, and Line 18 of Algorithm 1 is unlikely to add such solutions to the next pool.

Algorithm 1: the `migration` Algorithm

```
Input:
    $p$ is the project whose library is replaced with a newer version.
Output:
    $p'$ is the migrated project.
1: errors$\leftarrow$ compile(p); pool$\xleftarrow{add}$ p; nobetter$\leftarrow$ 0; generation$\leftarrow$ 0;
2: while errors $\neq \emptyset$ and nobetter<10 and generation<100 do
3:     nextpool$\leftarrow \emptyset$ ;
4:     for p $\in$ pool do
5:         errors$\leftarrow$ compile(p);
6:         for error $\in$ errors do
7:             op$\leftarrow$ operator(error); solution$\leftarrow$ apply(op, p);
8:             nextpool$\xleftarrow{add}$ solution;
9:         end for
10:    end for
11:    best$\leftarrow$ best(pool); nextbest$\leftarrow$ best(nextpool);
12:    if compile(nextbest)<compile(best) then
13:        errors$\leftarrow$compile(nextbest);
14:        nobetter$\leftarrow$0;$p'$ $\leftarrow$ nextbest;
15:    else
16:        nobetter$\leftarrow$nobetter+1;
17:    end if
18:    generation$\leftarrow$ generation+1; pool $\xleftarrow{top10}$ nextpool;
19: end while
20: $p'$ $\leftarrow$ best(pool);
```

4.1.4 Other issues. LibCatch implements several migration operators to handle other issues of updating client code.

MA12. Resolving ambiguous types. This migration operator is designed to resolve ambiguous types. If JDT reports that a type is ambiguous, *MA12* generates a precise `import` statement to resolve the problem. As LibCatch works in a try-and-validate manner, it does not have to determine which type is the correct type. Instead, it generates a solution for each ambiguous type. For example, the examples of `cassandra 1.2.0` import all the types of two packages:

```
import org.apache.cassandra.db.*;
import org.apache.cassandra.thrift.*;
```

As the later versions implement two `ConsistencyLevel` types under the above packages, JDT reports that the type is ambiguous. LibCatch replaces the above statements with:

```
import org.apache.cassandra.db.ConsistencyLevel;
```

LibCatch generates a solution for each possible type. In the other solution, it replaces the above statement with the `ConsistencyLevel` type under the `thrift` package. This solution is discarded, since it introduces more errors.

MA13. Replacing invisible fields with getters and setters. If an updated library hides visible fields, JDT reports that fields are invisible. *MA13* replaces such field accesses with getters and setters. Figure 1c presents an example.

MA14. Removing the `@Override` annotation. If a client-code method overrides an API method, this client-code method will be marked with a `@Override` annotation. If this API method is deleted, the annotation will cause a compilation error. To resolve the error, *MA14* removes the `@Override` annotation from the client-code method.

4.2 Guiding the Migration Process

As shown in Algorithm 1, LibCatch works in a try-and-validation manner. Line 2 stops when (1) a solution without compilation errors is synthesized; (2) no better solutions are found in the recent ten generations; or (3) it fails to resolve all the compilation errors within one hundred generations. For each solution in our pool, Line 5 compiles it to collect its compilation errors. An error location is a code element, which introduces a compilation error after API libraries are updated. LibCatch uses JDT [1] to extract compilation errors, since we focus on updating Java code and JDT is the built-in compiler of the Eclipse IDE. When JDT locates a compilation error, it reports the code range of the error. Based on each code range and its corresponding Abstract Syntax Tree (AST), LibCatch locates its code element with errors. Besides code ranges, JDT reports the types of compilation errors. For example, if it fails to resolve a type named `A`, JDT reports its error message: "`A` cannot be resolved to a type." From JDT, LibCatch extracts the types of compilation errors. As shown in Line 7, for each type of compilation error, LibCatch enumerates all feasible migration operators as defined in Section 4.1 to generate solutions. Line 8 adds new solutions to a new pool. Line 11 obtains the best solutions from both pools. The best solution has the fewest compilation errors. Line 12 compares the best solutions from the two pools. If the new pool does not contain a better solution, Line 16 increases `nobetter`. If it does, Lines 13 and 14 reset the variable and assign the better solution to p'. Line 18 increases `generation`, and selects the top ten unique solutions from the new pool. It determines that two solutions are duplicated, if their compilation errors are identical.

Ni *et al.* [60] propose an approach that migrates deep learning clients from calling TensorFlow to PyTorch. They use the messages of compilation errors to guide their migration process. LibCatch is built upon JDT [1]. JDT produces a unique error ID for each type of compilation errors. LibCatch uses the Ids to select operators, since they are more reliable than messages. In addition, JDT reports the code elements that trigger compilation errors. LibCatch applies selected migration operators to such code elements. For example, if an error Id is related to undefined methods, according to Table 2, LibCatch can select *MA1* and apply the change to the method call inside the code element that triggers the error.

Many other software engineering problems can be reduced to optimization problems [30], and similar algorithms have been widely used in various SE research topics such as generating test cases [68], planning release time [29], repairing bugs [54, 73], refactoring [61], and estimating development cost [35]. When repairing bugs [54, 73], researchers [65] criticize that some bugs are not fully fixed, due to various technical challenges. For example, one of the challenges lies in the difficulty to generate full test inputs and oracles [45]. Although our migration algorithm also works in a try-and-validation manner, our target problem is simpler. We do not take test cases as our inputs, and our approach is less suffered from the challenges of repairing bugs.

5 EVALUATION ON BENCHMARK

We conduct evaluations to explore the research questions:

(RQ1) How effective is LibCatch (Section 5.2)?
(RQ2) What are the differences between manual updates and those from LibCatch (Section 5.3)?
(RQ3) How effective are only API mappings (Section 5.4)?

Table 3: The subjects.

Name	FV	LV	TV	Class	Field	Method
accumulo	1.3.6	1.9.2	29	99	119	182
cassandra	0.8.0	3.11.2	162	82	57	63
karaf	1.6.0	4.2.3	72	19	7	25
lucene	1.9.0	7.4.0	92	169	173	107
poi	3.0	4.0.1	21	458	833	1,193

5.1 Benchmark and Measure

Table 3 shows the subjects to build our benchmark. We select these projects, since they provide the archives of all their versions and they provide API usage examples in their released source files. In this section, all the projects are collected from the Apache Foundation, but in Section 6 we select projects from Github to collect diverse subjects. Column "FV" lists the first versions. Column "LV" lists the last versions when we wrote the paper. Column "TV" lists the number of the total versions. We select their API usage examples to build our migration tasks. For these examples, Columns "Class", "Method", and "Field" show the number of unique class APIs, method APIs, and field APIs that are used by those examples.

The tasks of the prior benchmarks [24, 28] are derived from migration commits, but most commits have compilation errors that are unrelated to differing versions of libraries [69]. As such errors mislead our approach, we do not reuse the prior benchmarks [24, 28]. Still, even if we did not include their tasks, our migration tasks involve many more APIs than the combination of all prior benchmarks. For example, the migration tasks of Gao *et al.* [28] involve 13 APIs, and those of Fazzini *et al.* [24] involve 20 APIs. As migration commits are difficult to obtain, the prior benchmarks use only several pairs of library versions. As a comparison, our tasks involve hundreds of API classes, and more than one hundred version pairs. Section 5.2.1 introduces how to build our tasks.

To evaluate the correctness of migrated code, researchers calculate the compilation errors of migrated code [56, 80] or compare migration code with the results of humans [24, 28]. We followed the same practice and used both measures. In RQ1 and RQ3 of this section, we calculate the compilation errors of migrated code. In RQ2, we compare our fully migrated code with manual edits. In Section 6, we calculate the compilation errors of migrated projects, and we manually analyze the root causes of unsuccessful cases.

5.2 RQ1. Reduced Compilation Errors

5.2.1 Setup. When an API usage example is shipped with a library whose version is `v1`, the example must call the APIs that are declared by `v1`. For each example, we replace its library dependency from `v1` to the next version, `v2`, to build a migration task. If an example produces compilation errors after its library is replaced, we consider this example as a migration task. In Table 4, Column "Total" shows the total number of tasks. There are fewer tasks than the total combinations of version pairs, since some next versions do not introduce API-breaking changes or such changes are not called.

5.2.2 Result. Table 4 shows the result. Column "Total" lists the total tasks. Column "LibCatch" lists our migration results. As defined in the semantic versioning [2, 49], the first number of a version indicates that the library has API-breaking changes. We notice that this rule is do not apply in all libraries. For example,

Table 4: The migration results on our benchmark.

Name	Total	LibCatch							MA1 (Only mappings)						
		-	|	+	full	%	error	solution	-	|	+	full	%	error	solution
accumulo	24	23	1	0	11	45.8%	3.6	17.4	0	23	1	0	0.0%	-1.3	0.2
cassandra	72	71	1	0	43	59.7%	3.7	17.9	3	66	3	0	0.0%	0.0	0.2
karaf	3	2	1	0	1	33.3%	2.3	17.3	0	3	0	0	0.0%	0.0	0.0
lucene	13	9	4	0	2	15.4%	9.0	25.6	3	10	0	0	0.0%	1.5	0.6
poi	11	9	2	0	4	36.4%	13.5	41.7	1	9	1	0	0.0%	0.2	0.6
total	123	114	9	0	61	49.6%	5.1	20.7	7	111	5	0	0.0%	-0.1	0.3

Table 3 shows that the first number of `accumulo` does not change, but we builds 23 tasks from this library. In the contrast, as `accumulo` more strictly follows this rule, we build only three tasks from this library (`1.6.0->2.0.0`, `2.1.2->2.1.3`, and `4.2.2->4.2.3`), and two tasks are between major versions.

Subcolumns "-", "|", and "+" show tasks whose compilation errors are reduced, unchanged, and increased, respectively. LibCatch reduces the compilation errors for 95.1% (114/123) of tasks, without increasing such errors in any tasks. Subcolumns "full" shows the number of migrated tasks whose compilation errors are eliminated. Subcolumns "%" are calculated as $\frac{full}{Total}$. In total, LibCatch eliminates compilation errors in 49.6% of tasks. Subcolumns "error" show the averages of reduced compilation errors per task. Subcolumns "solution" show the averages of generated solutions per task.

As the next versions of libraries often have trivial changes, most tasks have few compilation errors. In 48 tasks, each task has only a compilation error. LibCatch fully fixed 45 of them. The other 16 fully fixed tasks have more compilation errors. Several tasks have hundreds of compilation errors. LibCatch did not fully fix them but reduced their compilation errors. For example, the `lucene4.6.1->lucene4.7.0` task has 141 compilation errors, and LibCatch reduced them to 73. Intuitively, a migration task becomes more challenging if two versions are more different. Open source communities have implemented tools to warn programmers of deprecated libraries. For example, Github has released Dependabot [75]. If Dependabot is enabled, it can submit pull requests if new versions of libraries are released. He *et al.* [33] report that Dependabot reduces the time of upgrading libraries. With such supports, programmes can update libraries between near versions, which are less challenging for tools like LibCatch.

In summary, our results show that LibCatch reduces the compilation errors in 95.1% of tasks, and 49.6% of tasks are fully resolved.

5.3 RQ2. Comparison with Manual Migration

5.3.1 Setup. In RQ1, 61 tasks were fully migrated. In this RQ, we manually inspected all these tasks. Each task illustrates a migration process, in which the library of an example is replaced with the next version. If programmers update API examples, we compare the edits of programmers with the edits of LibCatch. However, programmers can forget to update API usage examples. First, as early code repositories do not provide the support of continuous integration, their compilation errors are not identified. Second, as examples are used to illustrate API usage, they are shipped in the format of source files. As they are not included in compiled code, build scripts often exclude them from compilation. In these tasks, LibCatch eliminates compilation errors but programmers leave them unfixed. Here, we do not ask programmers to check our edits, since they are interested in checking bugs in the latest versions but most tasks are not built from the latest versions.

5.3.2 Result. Based on our inspection results, we classified the tasks into four categories:

1. In 24 tasks, our migrated projects are identical to manual migrations. For example, the task of `cassandra 1.0.0` has an error: "The method hexToBytes(String) is undefined for the type FBUtilities". LibCatch replaces the method call with `hexToBytes(String)` of the `ByteBufferUtil` type, which is identical to manual migrations.

2. In 33 tasks, programmers forget to update obsolete API examples, but LibCatch removes all their compilation errors. These API examples are confusing for client programmers, since they even do not compile. For example, in the example of `cassandra 0.8.1`, the `CassandraBulkLoader` class is as follows:

```
baseColumnFamily = new ColumnFamily(ColumnFamilyType.Standard,
        DatabaseDescriptor.getComparator(keyspace),
        DatabaseDescriptor.getSubComparator(keyspace, columnFamily
        ), CFMetaData.getId(keyspace, columnFamily));
```

As the `ColumnFamily` constructor is deleted since `0.8.1`, the above code produces a compilation error. The compilation error is not fixed from `cassandra 0.8.1` to `0.8.10`. Although programmers fix this error in later versions, we cannot find a manual reference that illustrates how to migrate the example from `0.8.0` to `0.8.1`. LibCatch replaces the constructor with a creator:

```
baseColumnFamily = ColumnFamily.create(CFMetaData.getId(
        keyspace, columnFamily));
```

3. In 2 tasks, programmers delete API examples. For example, the task of `cassandra 1.2.0-beta3` has a code line:

```
EnumSet<Permission> authorized = Permission.NONE;
```

As the type of `Permission.NONE` is modified to `Set<Permission>` in the later versions, the above code example produces a type mismatch. LibCatch adds an explicit cast to resolve it:

```
EnumSet<Permission> authorized = (EnumSet<Permission>)
        Permission.NONE;
```

Programmers delete this example from the later versions, so we cannot compare our modifications with manual migrations.

4. In 2 tasks, programmers modify library code. For example, in our task, the example of `lucene 4.0.0` has an error: "The constructor TextField(String, BufferedReader) is undefined", since its next version deletes this constructor. LibCatch modifies client code to call other constructors. The programmers of `lucene` revert the deletion to resolve the issue, but their API example is unchanged.

In summary, in 24 tasks, our migration edits are identical to manual modifications, and in 33 tasks, compilation errors are ignored by programmers but are resolved by LibCatch.

5.4 RQ3. Migrating with Only API Mapping

5.4.1 Setup. As mining API mappings is an important research line, it is interesting to explore the effectiveness of our approach, if it migrates code with only such mappings. In this research, we enable only MA1 of LibCatch to update code, and compare its results with those in RQ1.

5.4.2 Result. In Table 4, Column "MA1" lists the migration results of only mappings. Compared with Column "LibCatch", MA1 does not change the compilation errors for more than 90% of tasks, and for five tasks, it even increases errors. Although MA1 reduces the compilation errors of seven tasks, it fails to fully migrate any task. The results indicate the mappings alone are insufficient to migrate many complex cases. The next section makes a further comparison.

6 MIGRATION IN THE WILD

In this section, we use LibCatch and other tools to migrate real projects in the wild. LibCatch takes different inputs from the prior tools. As their inputs cannot be aligned, the comparison results do not indicate better techniques. Still, it illustrates the situations, when programmers use tools to migrate their code.

6.1 Compared Tools

As we introduced in Section 1, the prior approaches fall into two research lines. From each research line, we select a typical approach.

For mining API mappings, we select SemDiff [21], since it won the ACM SIGSOFT distinguished paper award and we are unaware of more recent approaches in this research line. SemDiff [21] recommends replacements for API methods. In this section, we compare LibCatch with SemDiff. As SemDiff does not include techniques to modify client code, we assume that it can resolve a compilation error, if it recommends a correct replacement.

For learning edit scripts, we select Meditor [78], since its project website [11] provides both its source code and the collected migration commits. We do not select APIFix [28], since it has an unfixed compilation error [5] and it does not released the collected migration commits. Meditor [78] infers edit scripts from migration commits, and it extracts migration commits from Github. As Github has a limit for retrieving its contents [8], retrieving migration commits is troublesome, and it takes much more time than inferring edit scripts. As Meditor releases their collected migration commits on their website, we can save the collection effort.

6.2 Setting

As the first approach of a new research line, our approach takes different inputs from the prior research lines. Our benchmark does have some inputs that are required by the prior approaches. For example, prior approaches [24, 28] need change examples to learn edit scripts, but we do not need such inputs. Although it is infeasible to conduct a fair controlled experiment, it is feasible to derive interesting findings, if these tools are evaluated in the wild. The results can be useful for understanding the benefits and pitfalls of migration tools in real development. In Section 6, we select two tools [21, 78] from the other research lines, and compared those tools with LibCatch in terms of migrating real projects.

In this section, we focus on `lucene`, since it is a popular library. Other researchers also select `lucene` in their studies. For example, the only overlapped subject between Xu *et al.* [78] and ours is `lucene`. All the subjects in Section 5 are collected from the Apache Foundation. As a supplement, all the subjects in this section are collected from Github. In particular, we select a project (p) if it satisfies the following five criteria: (1) p uses any of `lucene`'s versions lower than `7.4.0`; (2) p uses Maven as the build system, because Maven allows us to easily change the version information of any library dependency; (3) p can be successfully built without any compilation errors; (4) p is not a toy project according to the project's description, and we also require that p should be forked and have stars; and (5) p does not call `solr`. If the version of `solr` does not match the version of `lucene`, it causes compilation errors, but we cannot resolve compilation errors in the `jar` files of `solr`. As a result, we remove such projects. Table 5 shows our selected projects. In total, we selected 15 projects. Here, as we need to manually inspect whether the mappings from SemDiff are useful, we cannot afford selecting more projects. Except in three projects, compilation errors occur when the first number of the version changes.

Column "LOC" lists the lines of code. Column "EL" lists the versions of their existing `lucene`. Column "UL" lists the versions of the updated `lucene` library. For each project, we increase the version of its `lucene` library, until the breaking changes of that version cause compilation errors. We use this strategy to minimize the compilation errors in a task, since it takes too much manual effort to inspect the results of SemDif.

Column "Compilation error" lists the number of compilation errors, when `lucene` is updated to newer versions. The categories of compilation errors are defined in Table 2. Subcolumn "All" lists all compilation errors. As shown in Row "Total", first, 39.2% of compilation errors are caused by missing API elements. API mappings can resolve these errors. Second, 20.7% of compilation errors are caused by incompatible calls. These errors can be partially resolved by swapping method parameters. Finally, 16.5% of compilation errors need to add or remove API calls. These errors can be partially resolved by generating method stubs. The remaining 6.0% of compilation errors need other edits.

As we have no manually migrated references for these projects, in this RQ, we use the number of compilation errors to measure the quality of migrated projects. We released our migrated projects on our website, so other researchers can recheck the correctness of our migrated results.

6.3 Result

In Table 5, Columns "LibCatch", "SemDiff", and "Meditor" show the results of the three tools, respectively. Subcolumn "#" lists the number of total compilation errors in migrated projects, and Subcolumn "Δ" shows the number of reduced compilation errors. For SemDif, we manually try its replacements to determine whether it can resolve a compilation error. According to the results, we classify the projects into three categories:

Table 5: The migration results in the wild

Project	LOC	EL	UL	Compilation error					LibCatch		SemDiff		Meditor	
				ME	IE	MF	Other	All	#	Δ	#	Δ	#	Δ
explicit-semantic-analysis	1,146	4.8.1	5.0.0	15	6	2	0	23	0	23	23	0	23	0
flea-db	1,664	4.10.3	5.0.0	2	4	0	0	6	0	6	4	2	6	0
IKAnalyzer	2,509	4.10.0	5.0.0	4	1	1	1	7	0	7	7	0	7	0
stemmer-sk	303	5.3.1	6.3.0	1	1	0	0	2	0	2	2	0	2	0
lucene-layer	2,377	4.4.0	4.5.0	0	0	10	0	10	0	10	10	0	10	0
lucene-generic-highlighter	155	3.5.0	4.0.0	3	0	0	0	3	0	3	3	0	3	0
lucene-multilingual	488	3.6.0	4.0.0	9	0	0	0	9	0	9	9	0	9	0
lqt	1,222	6.1.0	7.0.0	4	1	0	0	5	1	4	5	0	5	0
greplin-lucene-utils	3,957	3.5.0	3.6.0	4	0	1	12	17	1	16	17	0	17	0
LuceneQueryExpansion	5,566	4.8.1	5.0.0	14	4	0	0	18	3	16	18	0	18	0
word2vec-query-expansion	986	4.6.1	5.0.0	7	2	2	0	11	1	10	11	0	11	0
lire	20,342	4.2.0	5.0.0	23	15	0	1	39	1	38	39	0	39	0
anserini	9,282	6.3.0	7.0.0	1	0	14	0	15	1	14	15	0	15	0
lucene-interval-fields	503	3.5.0	4.0.0	19	7	9	0	35	9	26	35	0	35	0
marple	1,579	6.5.1	7.0.0	29	8	0	0	37	21	16	37	0	37	0
Total	52,079	n/a	n/a	93	49	39	14	237	38	200	235	2	237	0

EL: the existing `lucene`; UL: the updated `lucene`; ME, IE, MF, and Other are defined in Table 2; #: the compilation errors in updated code; and Δ: the reduced compilation errors.

1. In 7 projects (46.7%), LibCatch resolves all the compilation errors; SemDiff can resolve the 2 compilation errors of only `flea-db`; and Meditor resolves no compilation errors. `flea-db` [22] is a database that supports the persistence of objects. It has 2,611 lines of code, and is built on `lucene 4.10.3`. In this migration task, we also replaced its `lucene` library with `5.0.0`. The replacement introduced 6 compilation errors, including 2 undefined methods and 4 incompatible parameters. LibCatch resolved all the six compilation errors. After inspecting its migration log, we find that MA6 resolved 3 compilation errors; MA8 resolved 2 compilation errors, and MA11 resolved 1 compilation error. For example, a modification is as follows:

```
-TopFieldCollector.create(this.sort, 1, null, false, false,
      false, false);
+TopFieldCollector.create(this.sort, 1, false, false, false,
      false);
```

In `5.0.0`, the third formal parameter of the `create` method is deleted. MA8 removed the corresponding actual parameter to solve the problem. `flea-db` implements 15 test methods, and our migrated code passed all its test cases. SemDiff located the correct replacement for two method calls. In particular, it found the replacement for the `create` method that is identical to LibCatch (the support is 1 and the confidence is 0.143). For the callsite of `setIndexed(boolean)`, LibCatch removed it with MA11, but SemDiff recommended replacing it with `setIndexOptions(IndexOptions)`. As its support is 41 and its confidence is 0.489, their recommended replacement can be more frequent than our removals.

As another example, `explicit-semantic-analysis` [70] leverages Wikipedia dumps to compare the semantic similarity between two given texts. It is a research tool, and its papers are published in the top AI venues [26, 27]. Google scholar reports that the two papers have more than 3,000 citations. The latest `explicit-semantic-analysis` has 1,514 lines of code, and is built on `lucene 4.8.1`. In this migration task, we replaced its `lucene` library from `4.8.1` to `5.0.0`. After that, `explicit-semantic-analysis` produced 23 compilation errors. LibCatch migrated `explicit-semantic-analysis` resolved all the compilation errors. After inspecting the migration log, we find that MA6 resolved 16 compilation errors. For example, MA6 made a modification as follows:

```
-queryParser=new QueryParser(LUCENE_48, TEXT_FIELD, analyzer);
+queryParser=new QueryParser(TEXT_FIELD, analyzer);
```

`lucene 5.0.0` also removes the first parameter of the `QueryParser` constructor. MA6 removed the actual parameter of the old version to resolve this compilation error. MA8 resolved 5 compilation errors, and one is as follows:

```
 -... = FSDirectory.open(termDocIndexDirectory)){
 +... = FSDirectory.open(termDocIndexDirectory.toPath()){
```

In the above patch, the type of `termDocIndexDirectory` was `File`, but the type of the formal parameter was changed to `Path`. MA8 resolved the problem by calling the `toPath()` method of the actual parameter. In addition, MA9 generated stubs to resolve the remaining 2 unimplemented methods. `explicit-semantic-analysis` has two test methods, and our migrated code passed both test methods.

The 23 compilation errors include 9 unresolved variables, 6 undefined constructors or methods, 6 incompatible parameters, and 2 unimplemented methods. SemDiff has the potential to resolve the 6 undefined constructors and methods. However, it does not find the replacement for the constructor, and the remaining 5 undefined methods require adding method calls than replacements. As result, SemDiff resolves no compilation errors.

Meditor fails to resolve any compilation errors. As the combination of versions are many, the migration edits for a specific pair of versions are few. In total, Xu *et al.* [78] extracted 322 migration commits of `lucene`, but even the most frequent `lucene` version pair (`3.0.2-4.0`) has only 24 migration commits. Table 2 of Xu *et al.* [78] presents the top ten release pairs of Lucene. We compare them with the release pairs listed in Table 5 of our paper, but we find no overlap. Their tenth release pair has only 8 migration commits, and other pairs have even fewer commits. From all their migration commits, they infer 153 edit scripts, but even if their inference is fully accurate, their inferred scripts cannot migrate code of other version pairs. For example, if programmers need to update the callsites of `lucene` `3.0.2` to other versions, the migration commits from `3.0.2` to `4.0` are less useful, since other versions declare quite different APIs.

2. In 6 projects (40.0%), LibCatch resolved most compilation errors, but the other two tools failed to resolve any compilation errors. In `flea-db` and `explicit-semantic-analysis`, we introduce the cases where both tools successfully resolve compilation errors. Here, although our improvements over the other two tools are significant, for the benefits of follow-up researchers, we analyze the cases where both tools fail to resolve compilation errors. For example, `greplin-lucene-utils` has the following code:

```
1 public class ForwardingIndexReader extends IndexReader {
2   private final IndexReader delegate;
3 @Override
4 public ... getFieldNames(final FieldInfos fldOption) {
5   return this.delegate.getFieldNames(fldOption);
6 }}
```

After we update the `lucene` library from `3.5.0` to `3.6.0`, Line 3 of the above code reports a compilation error, since the `getFieldNames` method is removed from `IndexReader`. After *MA8* removed the `@Override` annotation, this compilation error is resolved, but Line 5 still calls the deleted method. LibCatch does not resolve this issue. SemDiff does not recommend any replacements for this method either. We manually inspect its documents and implementations, but find no replacement. Instead, we find a message in the parameter type `FieldInfos`:"WARNING: This API is experimental and might change in incompatible ways in the next release." According to this warning, `lucene` may not implement replacements for `getFieldNames`, and even `FieldInfos` can be deleted in future versions. We try to delete Lines 4 to 6 from the above code. After the deletion, the compilation error is resolved, since this method is not called by other code locations.

As another example, `LuceneQueryExpansion` has the following code:

```
Directory dir = DirectoryReader.open(index);
```

Between `lucene` `4.8.1` and `5.0.0`, both the input type and the output type of `open` are changed. A feasible migration can be as follows:

```
Directory dir = DirectoryReader.open(new SimpleFSDirectory(
        index.toPath())).directory();
```

MA8 has the potential to generate the above code. However, it needs to apply at least three *MA8* edits, but the guidance algorithm is not sufficiently smart to make the desirable combination.

SemDiff does not find replacements for this above API call, and this error cannot be resolved by only replacements. For this example, Meditor finds no migration commit from `4.8.1` to `5.0.0`. In total, it finds 11 migration commits whose source version is `4.8.1`, but these commits either migrate other APIs or do not modify source files. For example, one of their collected migration commits is from `4.8.1` to `4.10.3` [9], but this commit does not modify source files. As another example, a migration commit is from `4.8.1` to `5.5.4` [10], but the source version of this commit does not call the relevant methods (*e.g.*, `DirectoryReader.open()`).

In Table 5, Subcolumn "ME" lists the compilation errors caused by missing API elements. To resolve ME errors, a tool needs the mappings of API classes, methods, and fields, but many approaches do not mine all mappings. For example, SemDiff mines the mappings of only API methods, and other approaches [56, 80] mine mappings for API classes. While other approaches [17, 72] can mine more mappings than SemDiff, their mined mappings can resolve only the compilation errors in this subcolumn. As a result, they will resolve fewer compilation errors than LibCatch (93 vs 200). It is also unlikely that other tools can collect more useful migration commits. For example, APIFix recommends to retrieves the clients of a library through the Github dependency graph [6], but like many other libraries, `lucene` does not list its dependents [7]. Without migration commits, even if a tool (*e.g.*, APIFix) can have more advanced inference techniques than Meditor, it is unlikely to resolve more compilation errors. Furthermore, programmers often need to update their libraries to the latest versions, but the migration commits whose target versions are the latest versions are even fewer than other combinations of versions.

In summary, LibCatch resolves most compilation errors in 13 out of our 15 projects; SemDiff resolves only two undefined methods; and Meditor resolves no compilation errors. We further discuss our unresolved compilation errors in Section 8.

7 THREATS TO VALIDITY

The threats to external validity include our limited subjects. To reduce the threat, we select projects from both Apache and Github. However, all our migration operators are designed for Java. The threat can be mitigated by selecting subjects from more sources. The internal threat to validity includes wrong manual labels. We use manual migration edits as the true labels, but programmers can forget to update their API examples. As a result, LibCatch can be more effective than what we calculated. The problem can be mitigated by migrating client code from more reliable sources.

8 CONCLUSION AND FUTURE WORK

The prior research directions mine API mappings or learn edit scripts, but they have inherent limitations. To resolve their limitations, we introduce a new research line that migrates clients based on the feedback of compilers. As the first exploration, we propose LibCatch, and prepare 123 migration tasks in our benchmark. Among them, LibCatch resolved all the compilation errors in 61 tasks, and produced edits that are identical to manual migrations in 33 tasks. We evaluated LibCatch in the wild. In 13 out of 15 real projects, it reduced compilation errors to no more than two. Our first exploration achieves promising results, and some potential research opportunities are as follows:

Tuning LibCatch. Tuning LibCatch can further improve its effectiveness. For example, (1) MA1 uses simple techniques to mine API mappings, and advanced techniques can be useful. Some complicated cases require many-to-many mappings, but it is still an open question to mine such mappings [44, 80]. (2) MA2 selects only the top ten candidates, but more solutions can be generated if we select more candidates. (3) MA4 searches only with "create", and other words can retrieve more useful candidates; (4) MA6 can implement other merging strategies; and (5) MA10 can implement more advanced handling techniques. As future libraries can introduce unexpected changes to APIs, it is an endless task to explore a complete set of migration operators. Migration operators can be derived from various sources (*e.g.*, revision histories [24, 78] and library changes [52]). For migration algorithms, researchers [23, 48] proposed various algorithms to resolve optimization problems. As the quests for both migration operators and optimization techniques are endless [25], our direction still has great research potential.

Migrating code in other languages and cross languages. Although our tool is implemented to handle only Java code, our idea can work on other languages and even cross languages, since the failures of their API migrations cause compilation errors and such errors provide valuable hints to migrate code. Still, such failures can cause compilation errors that rarely appear in Java code. More migration operators are required to handle such errors.

Detecting bugs in migrated code. Researchers [64, 65] show that it is feasible to cheat an automatic measure. API changes can introduce behavioral differences [18, 55, 79], but our fitness function does not consider bugs caused by such differences. After we resolve all compilation errors, it becomes feasible to detect such bugs with testing techniques. Our overhead is less than automatic program repair (APR), since we do not need the time to execute test cases but we need the time to compile and modify code like APR does.

ACKNOWLEDGEMENT

We appreciate reviewers for their insightful comments. Hao Zhong is sponsored by National Nature Science Foundation of China No. 62272295. Na Meng is sponsored by NSF-1845446.

REFERENCES

[1] 2019. JDT. http://www.eclipse.org/jdt/.
[2] 2019. Semantic versioning. https://semver.org/.
[3] 2022. Apache Log4j security vulnerabilities. https://logging.apache.org/log4j/2.x/security.html.
[4] 2022. Apache Log4j vulnerability guidance. https://www.cisa.gov/uscert/apache-log4j-vulnerability-guidance.
[5] 2023. APIFix bug. https://github.com/gaoxiang9430/APIFix/issues/1.
[6] 2023. APIFix miner. https://github.com/gaoxiang9430/APIFix/blob/master/doc/miner.md.
[7] 2023. The dependents of lucene. https://github.com/apache/lucene/network/dependents.
[8] 2023. Github limit. https://docs.github.com/en/apps/creating-github-apps/registering-a-github-app/rate-limits-for-github-apps.
[9] 2023. lucene update of gora. https://github.com/apache/gora/commit/29cbf8ab03eb68898db0014e416de27d4932231d.
[10] 2023. lucene update of maven-indexer. https://github.com/apache/maven-indexer/commit/532ea64eecd499d047bf4211b6a5bde41f1a7c72.
[11] 2023. Meditor. https://bitbucket.org/shengzhex_research/meditor/.
[12] Jesper Andersen and Julia L Lawall. 2010. Generic patch inference. *Automated software engineering* 17, 2 (2010), 119–148.
[13] I. Balaban, F. Tip, and R. Fuhrer. 2005. Refactoring support for class library migration. In *Proc. OOPSLA*. 265–279.
[14] Anatole Beck. 1964. On the linear search problem. *Israel Journal of Mathematics* 2, 4 (1964), 221–228.
[15] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. 2020. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering* 25, 2 (2020), 1458–1492.
[16] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *Proc. SANER*. 507–511.
[17] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2019. Mining likely analogical APIs across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering* 47, 3 (2019), 432–447.
[18] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proc. ICSE*. 112–124.
[19] Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In *Proc. ICSM*. 359–368.
[20] Bradley E Cossette and Robert J Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proc. ESEC/FSE*. 1–11.
[21] Barthélémy Dagenais and Martin P Robillard. 2011. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology* 20, 4 (2011), 1–35.
[22] Ignacio del Valle Alles. 2019. org.brutusin:flea-db. https://github.com/brutusin/flea-db.
[23] Wu Deng, Junjie Xu, Yingjie Song, and Huimin Zhao. 2021. Differential evolution algorithm with wavelet basis function and optimal mutation strategy for complex optimization problem. *Applied Soft Computing* 100 (2021), 106724.
[24] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proc. ISSTA*.
[25] Dimitris Fouskakis and David Draper. 2002. Stochastic optimization: a review. *International Statistical Review* 70, 3 (2002), 315–349.
[26] Evgeniy Gabrilovich and Shaul Markovitch. 2006. Overcoming the brittleness bottleneck using Wikipedia: Enhancing text categorization with encyclopedic knowledge. In *Proc. AAAI*, Vol. 6. 1301–1306.
[27] Evgeniy Gabrilovich, Shaul Markovitch, et al. 2007. Computing semantic relatedness using wikipedia-based explicit semantic analysis.. In *Proc. IJcAI*, Vol. 7. 1606–1611.
[28] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: output-oriented program synthesis for combating breaking changes in libraries. In *Proc. OOPSLA*. 1–27.
[29] Des Greer and Guenther Ruhe. 2004. Software release planning: an evolutionary and iterative approach. *Information and software technology* 46, 4 (2004), 243–253.
[30] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *Comput. Surveys* 45, 1 (2012), 11.
[31] Stefanus A Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automatic Android deprecated-API usage update by learning from single updated example. In *Proc. ICPC*. 401–405.
[32] Stefanus A Haryono, Ferdian Thung, David Lo, Julia Lawall, and Lingxiao Jiang. 2021. MLCatchUp: Automated Update of Deprecated Machine-Learning APIs in Python. In *Proc. ICSME*. 584–588.
[33] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering* (2023).
[34] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proc. ICSE*. 274–283.
[35] Sun-Jen Huang and Nan-Hsing Chiu. 2006. Optimization of analogy weights by genetic algorithm for software effort estimation. *Information and software technology* 48, 11 (2006), 1034–1045.
[36] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break–an empirical study. *Information and Software Technology* 65 (2015), 129–146.
[37] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *Proc. ASE*. 255–266.
[38] Sukrit Kalra, Ayush Goel, Dhriti Khanna, Mohan Dhawan, Subodh Sharma, and Rahul Purandare. 2016. POLLUX: Safely Upgrading Dependent Application Libraries. In *Proc. ESEC/FSE*. 290–300.
[39] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type migration in ultra-large-scale codebases. In *Proc. ICSE*. 1142–1153.
[40] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and Applying Type Changes. In *Proc. ICSE*.
[41] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. Understanding type changes in java. In *Proc. ESEC/FSE*. 629–641.
[42] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.
[43] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.
[44] Maxime Lamothe and Weiyi Shang. 2018. Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android. In *Proc. MSR*. 503–514.
[45] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software quality journal* 21 (2013), 421–443.
[46] Mingwei Liu, Yanjun Yang, Yiling Lou, Xin Peng, Zhong Zhou, Xueying Du, and Tianyong Yang. 2023. Recommending Analogical APIs via Knowledge Graph Embedding. In *Proc. ESEC/FSE*. 1496–1508.
[47] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proc. ESEC/FSE*. 727–739.
[48] Jayanta Mandi, Peter J Stuckey, Tias Guns, et al. 2020. Smart predict-and-optimize for hard combinatorial optimization problems. In *Proc. AAAI*, Vol. 34. 1603–1610.

[49] Stephen McCamant and Michael D Ernst. 2004. Early identification of incompatibilities in multi-component upgrades. In *In Proc. ECOOP*. 440–464.

[50] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: generating program transformations from an example. In *Proc. PLDI*. 329–342.

[51] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proc. ICSE*. 502–511.

[52] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A history-based matching approach to identification of framework evolution. In *Proc. ICSE*. 353–363.

[53] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *Proc. ESEC/FSE*. 925–936.

[54] Martin Monperrus. 2018. Automatic software repair: a bibliography. *Comput. Surveys* 51, 1 (2018), 1–24.

[55] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proc. ISSTA*. 215–225.

[56] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proc. ASE*. 457–468.

[57] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *Proc. ICSE*, Vol. 1. 858–868.

[58] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proc. ESEC/FSE*. 651–654.

[59] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proc. ICSE*. 819–830.

[60] Ansong Ni, Daniel Ramos, Aidan ZH Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2021. Soar: a synthesis approach for data science API refactoring. In *Proc. ICSE*. 112–124.

[61] Mark O Keeffe and Mel O Cinneide. 2008. Search-based refactoring for software maintenance. *Journal of Systems and Software* 81, 4 (2008), 502–516.

[62] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. 2022. Towards automated library migrations with error prone and refaster. In *Proc. SAC*. 1598–1606.

[63] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179.

[64] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair.. In *Proc. 36th ICSE*. 254–265.

[65] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proc. ISSTA*. to appear.

[66] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proc. ICSE*. 404–415.

[67] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2016. On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In *Proc. ICSME*. 400–410.

[68] Praveen Ranjan Srivastava and Tai-hoon Kim. 2009. Application of genetic algorithm in software testing. *International Journal of software Engineering and its Applications* 3, 4 (2009), 87–96.

[69] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017).

[70] Philip van Oosten. 2019. ESA. https://github.com/pvoosten/explicit-semantic-analysis.

[71] Louis Wasserman. 2013. Scalable, example-based refactorings with refaster. In *Proc. WRT*. 25–28.

[72] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. Clear: contrastive learning for API recommendation. In *Proc. ICSE*. 376–387.

[73] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proc. ICSE*. 364–374.

[74] Wei Wu, Yann Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. Aura: a hybrid approach to identify framework evolution. In *Proc. ICSE*. 325–334.

[75] Marvin Wyrich, Raoul Ghit, Tobias Haller, and Christian Müller. 2021. Bots don't mind waiting, do they? Comparing the interaction with automatically and manually created pull requests. In *Proc. BotSE*. 6–10.

[76] Zhenchang Xing and Eleni Stroulia. 2007. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (2007), 818–836.

[77] Zhenchang Xing and Eleni Stroulia. 2007. Differencing logical UML models. *Automated Software Engineering* 14, 2 (2007), 215–259.

[78] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *Proc. ICPC*. 335–346.

[79] Hao Zhong, Suresh Thummalapenta, and Tao Xie. 2013. Exposing behavioral differences in cross-language API mapping relations. In *Proc. ETAPS/FASE*. 130–145.

[80] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proc. 32nd ICSE*. 195–204.

[81] Hao Zhong, Xiaoyin Wang, and Hong Mei. 2022. Inferring bug signatures to detect real bugs. *IEEE Transactions on Software Engineering* 48, 2 (2022), 571–584.