

# Providing Upgrade Plans for Third-party Libraries: A Recommender System using Migration Graphs

Riccardo Rubei · Davide Di Ruscio · Claudio Di Sipio · Juri Di Rocco ·  
Phuong T. Nguyen

the date of receipt and acceptance should be inserted later

**Abstract** During the development of a software project, developers often need to upgrade third-party libraries (TPLs), aiming to keep their code up-to-date with the newest functionalities offered by the used libraries. In most cases, upgrading used TPLs is a complex and error-prone activity that must be carefully carried out to limit the ripple effects on the software project that depends on the libraries being upgraded. In this paper, we propose EvoPlan as a novel approach to the recommendation of different upgrade plans given a pair of library-version as input. In particular, among the different paths that can be possibly followed to upgrade the current library version to the desired updated one, EvoPlan is able to suggest the plan that can potentially minimize the efforts being needed to migrate the code of the clients from the library's current release to the target one. The approach has been evaluated on a curated dataset using conventional metrics used in Information Retrieval, i.e., precision, recall, and F-measure. The experimental results show that EvoPlan obtains an encouraging prediction performance considering two

different criteria in the plan specification, i.e., the popularity of migration paths and the number of open and closed issues in GitHub for those projects that have already followed the recommended migration paths.

## 1 Introduction

When dealing with certain coding tasks, developers usually make use of third-party libraries (TPLs) that provide the desired functionalities. Third-party libraries offer a wide range of operations, e.g., database management, file utilities, Website connection, to name a few. Their reuse allows developers to exploit a well-founded infrastructure, without reinventing the wheel, which eventually helps save time as well as increase productivity. However, TPLs evolve over the course of time, and API functions can either be added or removed, aiming to make the library become more efficient/effective, as well as to fix security issues. Upgrading clients' code from a library release to a newer one can be a daunting and time-consuming task, especially when the APIs being upgraded introduce breaking changes that make the client fail to compile or introduce behavioral changes into it [27]. Thus, managing TPLs and keeping them up-to-date becomes a critical practice to minimize the technical debt [3].

In order to upgrade a client  $C$  from a starting library version  $l_{v_i}$  to a target one  $l_{v_t}$ , the developer needs to understand both versions' documentation deeply, as well as to choose the right matching between corresponding methods. Things become even more complicated when several subsequent versions of the library of interest  $l$  have been released from  $v_i$  to  $v_t$ . In such cases, developers who want to reduce the technical debts, which have been accumulated due to libraries that have not been

---

Riccardo Rubei  
Università degli studi dell'Aquila, Italy  
E-mail: riccardo.rubei@graduate.univaq.it

✉ Davide Di Ruscio  
Università degli studi dell'Aquila, Italy  
E-mail: davide.diruscio@univaq.it

Claudio Di Sipio  
Università degli studi dell'Aquila, Italy  
E-mail: claudio.disipio@graduate.univaq.it

Juri Di Rocco  
Università degli studi dell'Aquila, Italy  
E-mail: juri.dirocco@univaq.it

Phuong T. Nguyen  
Università degli studi dell'Aquila, Italy  
E-mail: phuong.nguyen@univaq.it

upgraded yet, have first to decide the *upgrade plan* that has to be applied, i.e., how to go from  $l_{v_i}$  to  $l_{v_t}$  since many possible paths might be followed. In such cases, it is essential to have proper machinery to assist developers in choosing suitable upgrade plans to potentially reduce the efforts that are needed to migrate the client project  $C$  under development. It is possible to minimize migration efforts by identifying upgrade plans, which similar projects have already performed, and thus, by relying on the experiences of already upgraded clients. In this way, developers have the availability of supporting material, e.g., documentation, and snippets of code examples that can be exploited during the migration phases.

In the context of open-source software, developing new systems by reusing existing components raises relevant challenges in: (i) searching for relevant modules; and (ii) adapting the selected components to meet some pre-defined requirements. To this end, recommender systems in software engineering have been developed to support developers in their daily tasks [23, 8]. Such systems have gained traction in recent years as they are able to provide developers with a wide range of useful items, including code snippets [20], tags/topics [7, 9], third-party libraries [19], documentation [22, 24], to mention but a few. In the CROSSMINER project [8], we conceptualized various techniques and tools for extracting knowledge from open source components to provide tailored recommendations to developers, helping them complete their current development tasks.

In this work, we propose EvoPlan, a recommender system to provide upgrade plans for TPLs. By exploiting the experience of other projects that have already performed similar upgrades and migrations, EvoPlan recommends the plan that should be considered to upgrade from the current library version to the desired one. A graph-based representation is inferred by analyzing GitHub repositories and their *pom.xml* files. During this phase, EvoPlan assigns weights representing the number of client projects that have already performed a specific upgrade. Afterwards, the system employs a shortest-path algorithm to minimize the number of upgrade steps considering such weights. It eventually retrieves multiple upgrade plans to the user with the target version as well as all the intermediate passages.

To the best of our knowledge, there exist no tools that provide this type of recommendations. Thus, we cannot compare EvoPlan with any baselines but evaluate it by using metrics commonly used in information retrieval applications, i.e., precision, recall, and F-measure. Furthermore, we also evaluate the correlation between GitHub<sup>1</sup> issues data and the suggested

upgrade plans. In this sense, our work has the following contributions:

- *Gathering and storing of migration data*: Using NEO4J Java Driver,<sup>2</sup> EvoPlan stores the extracted data in a persistent and flexible data structure;
- *Recommendation of an upgrade plan list*: Considering the number of clients, EvoPlan suggests the most common upgrade plans that are compliant with those that have been accepted by the developers community at large;
- *Modularity and flexible architecture*: The proposed system can be seen as both an external module integrable into other approaches and a completely stand-alone tool that can be customized by end users;
- *Automated evaluation and replication package availability*: The performance of EvoPlan has been evaluated by employing the widely used ten-fold cross-validation technique. Last but not least, we make the EvoPlan replication package available online to facilitate future research.<sup>3</sup>

The paper is structured as follows. Section 2 presents a motivating example and existing migration tools in the literature. Furthermore, in this section we also highlight the open challenges in the domain. Section 3 introduces EvoPlan, the proposed approach to the recommendation of third-party library upgrades. In Section 4, we present the performed evaluation process. The results obtained from the empirical evaluation are presented in Section 5 together with possible threats to validity. The related work is reviewed in Section 6. Finally, we conclude the paper and envisage future work in Section 7.

## 2 Motivations and Background

TPLs offer several tailored functionalities, and invoking them allows developers to make use of a well-founded infrastructure, without needing to re-implementing from scratch [19]. Eventually, this helps save time as well as increase productivity. However, as libraries evolve over the course of time, it is necessary to have a proper plan to migrate them once they have been updated. So far, various attempts have been made to tackle this issue. In this section, we introduce two motivating examples, and recall some notable relevant work as a base for further presentation.

<sup>1</sup> <https://github.com/>

<sup>2</sup> <https://github.com/neo4j/neo4j-java-driver>

<sup>3</sup> <https://github.com/MDEGroup/EvoPlan>

## 2.1 Explanatory examples

This section discusses two real-world situations that developers must cope with during the TLPs migration task, i.e., code refactoring and vulnerable dependencies handling. In the first place, it is essential to consider different TPL releases that are conformed to the semantic versioning format.<sup>4</sup> A standard version string follows the pattern *X.Y*, in which *X* represents the *major* release and *Y* represents the *minor* one. Sometimes, releases can include a *patch* version *Z*, resulting in the final string *X.Y.Z*. We present an explanatory example related to *log4j*,<sup>5</sup> a widely used Java logging library. When it is upgraded from version *1.2* to version *1.3*, as shown in Listing 1 and Listing 2, respectively, a lot of internal changes happened which need to be carefully documented.<sup>6</sup> As it can be noticed, the main change affects the **Category** class which is replaced by the **Logger** class. Furthermore, all the former methods that were used by the deprecated class cause several failures at the source code level. For instance, the `setPriority` method is replaced by `setLevel` in the new version.

```

1 Category root = Category.getRoot();
2 root.debug("hello");
3
4 Category cat = Category.getInstance(Some.class);
5 cat.debug("hello");
6
7 cat.setPriority(Priority.INFO);

```

**Listing 1** *log4j* version 1.2.

```

1 Logger root = Logger.getRootLogger();
2 root.debug("hello");
3
4 Logger logger = Logger.getLogger(Some.class);
5 logger.debug("hello");
6
7 logger.setLevel(LEVEL.INFO);

```

**Listing 2** *log4j* version 1.3.

Though this is a very limited use case, it suggests that the code refactoring that takes place during the migration is an error-prone activity even for a single minor upgrade, i.e., from version *1.2* to version *1.3*. Additionally, the complexity dramatically grows in the case of a major release as it typically requires extra efforts rather than a minor one which are not welcome by the majority of developers [14]. Considering such context, the reduction of the time needed for a single migration step, even a minor one, is expected to improve the overall development process.

|                               |  |            |
|-------------------------------|--|------------|
| Dependency                    | Version                                      |            |
| <b>log4j:log4j</b>            | <div>&gt;= 1.2</div> <div>&lt;= 1.2.27</div> |            |
| Defined in                    | pom.xml                                      |            |
| Vulnerabilities               |  |            |
| Dependency                    | Version                                      | Upgrade to |
| <b>com.google.guava:guava</b> | <div>&gt; 11.0</div> <div>&lt; 24.1.1</div>  | ~> 24.1.1  |
| Defined in                    | pom.xml                                      |            |
| Vulnerabilities               | CVE-2018-10237 Moderate severity             |            |

**Fig. 1** GitHub Dependabot alert.

Concerning vulnerable dependencies, GitHub Dependabot<sup>7</sup> provides weekly security alert digests that highlight possible security issues for outdated dependencies of a repository, which can be of different languages, e.g., Python, Java, JavaScript.<sup>8</sup> An example of a Dependabot report is shown in Fig. 1.

As shown in Fig. 1, Dependabot suggests possible TPL upgrades to solve vulnerabilities in the given project. For instance, the **guava** dependency seems to be outdated, and thus the system automatically suggests jumping to the latest version, i.e., *24.1.1*. Though this alert can raise awareness of this evolution, it does not offer any concrete recommendations on how to perform the actual migration steps. In some cases, the bot does not provide any recommended version to update the project, e.g., for the **log4j** dependence. In this respect, we see that there is an urgent need for providing recommendations of the most suitable plan, so as to upgrade the library, as this can significantly reduce the migration effort.

## 2.2 Existing techniques

This section reviews some relevant work that copes with the migration problem.

Meditor [30] is a tool aiming to identify migration-related (MR) changes within commits and map them at the level of source code with a syntactic program differencing algorithm. To this end, the tool mines GitHub projects searching for MR updates in the *pom.xml* file and check their consistency with the WALA framework.<sup>9</sup>

<sup>4</sup> <https://semver.org/>

<sup>5</sup> <https://logging.apache.org/log4j/>

<sup>6</sup> <http://articles.qos.ch/preparingFor13.html>

<sup>7</sup> <https://dependabot.com/blog/github-security-alerts/>

<sup>8</sup> <https://dependabot.com/#languages>

<sup>9</sup> <https://github.com/wala/WALA>

**Table 1** Main features of TLPs migration systems.

| System             | Inferring migration | Incremental plan | Popularity | GitHub issues | Upgrading | Replacement | Applying Migration |
|--------------------|---------------------|------------------|------------|---------------|-----------|-------------|--------------------|
| Meditor [30]       | ✓                   | ✗                | ✓          | ✗             | ✓         | ✓           | ✓                  |
| Apiwave [11]       | ✓                   | ✗                | ✓          | ✗             | ✗         | ✓           | ✗                  |
| Graph Mining [26]  | ✓                   | ✗                | ✓          | ✗             | ✗         | ✓           | ✗                  |
| RAPIM [2]          | ✓                   | ✗                | ✓          | ✗             | ✗         | ✓           | ✓                  |
| Diff-CatchUp [29]  | ✓                   | ✗                | ✓          | ✗             | ✓         | ✓           | ✗                  |
| M <sup>3</sup> [4] | ✓                   | ✗                | ✗          | ✗             | ✗         | ✓           | ✓                  |
| <b>EvoPlan</b>     | ✓                   | ✓                | ✓          | ✓             | ✓         | ✗           | ✗                  |

Hora and Valente propose Apiwave [11], a system that excerpts information about libraries' popularity directly from mined GitHub project's history. Afterwards, it can measure the popularity of a certain TLP by considering the import statement removal or addition.

Teyton *et al.* [26] propose an approach that discovers migrations among different TLPs and stores them in a graph format. A token-based filter is applied on *pom.xml* files to extract the name and the version of the library from the artifactid tag. The approach eventually exhibits four different visual patterns that consider both ingoing and outgoing edges to highlight the most popular target.

RAPIM [2] employs a tailored machine learning model to identify and recommend API mappings learned from previously migration changes. Given two TPLs as input, RAPIM extracts valuable method descriptions from their documentation using text engineering techniques and encode them in feature vectors to enable the underpinning machine learning model.

Diff-CatchUp [29] has been conceived with the aim of proposing usage examples to support the migration of reusable software components. The tool makes use of the UMLDiff algorithm [28] to identify all relevant source code refactorings. Then, a heuristic approach is adopted to investigate the design-model of the evolved component and retrieve a customizable ranked list of suggestions.

Collie *et al.* recently proposed the M<sup>3</sup> tool [4] to support a semantic-based migration of C libraries. To this end, the system synthesizes a behavioral model of the input project by relying on the LLVM intermediate representation.<sup>10</sup> Given a pair of source and target TLPs,

the tool generates abstract patterns that are used to perform the actual migration.

Table 1 summarizes the features of the above-mentioned approaches by considering the different tasks involved in migration processes by starting with the discovery of possible migration changes up to embedding them directly into the source code as explained below.

- *Inferring migration*: To extract migration-related information, tools can analyze existing projects' artifacts, i.e., commits, *pom.xml* file, or tree diff. This is the first step of the whole migration process.
- *Incremental plan*: The majority of the existing approaches perform the migration just by considering the latest version of a TLP. This could increase the overall effort needed to perform the actual migration, i.e., developers suffer from accumulated technical debt. In contrast, considering a sequence of intermediate migration steps before going to the final one can reduce such refactoring.
- *Popularity*: This is the number of client projects that make use of a certain library. In other words, if a TLP appears in the *pom.xml* file or in the import statement, its popularity is increased.
- *GitHub issues*: As an additional criterion, the migration process can include data from *GitHub issues* that may include relevant information about TLPs migration. Thus, we consider them as a possible source of migration-related knowledge.
- *Upgrading*: This feature means that the tool supports the upgrading of a TLP from an older version to a newer one. For instance, the migration described in Section 2.1 falls under this class of migration.
- *Replacement*: Differently from upgrading, replacement involves the migration from a library to a different one that exposes the same functionalities.
- *Applying migration*: It represents the final step of the migration process in which the inferred migration changes are actually integrated into the project.

### 2.3 Dimensions to be further explored

Even though several approaches successfully cope with TPL migration, there are still some development dimensions that need to be further explored. However, providing an exhaustive analysis is out of the scope of this section. Thus, we limit ourselves to identify some of them by carefully investigating the approaches summarized in Table 1. The elicited dimensions are the following ones:

- *D1: Upgrading the same library*. Almost all of the presented approaches apart from Meditor, focus on

<sup>10</sup> <https://llvm.org/>

replacing libraries and very few support the upgrades of already included ones (see columns *Upgrading* and *Replacement* in Table 1).

- *D2: Varying the migration data sources.* During the inferring migration phase, strategies to obtain migration-related data play a fundamental role in the overall process. A crucial challenge should be investigating new sources of information besides the well-known sources e.g., Bug reports, Stack Overflow posts, and GitHub issues.
- *D3: Aggregating different concepts.* The entire migration process is a complex task and involves notions belonging to different domains. For instance, GitHub issues could play a relevant role in the migration process. A recent work [17] shows that the more comments are included in the source code, the lesser is the time needed to solve an issue. Neil *et al.* [18] extracted security vulnerabilities from issues and bug reports that could affect library dependencies.
- *D4: Identification of the upgrade plan.* Existing approaches identify and apply migrations by taking as input the explicit specification of the target version of the library that has to be upgraded. Providing developers with insights about candidate upgrade plans that might reduce the migration efforts can represent valuable support to the overall upgrade process.

In the present work we aim to explore and propose solutions for the dimensions D1 and D4 by providing multiple possible upgrade plans given the request of upgrading a given library to target a specific target version. Furthermore, we also perform an initial investigation on the D2 and D3 dimensions, relying on GitHub issues. As it can be seen in Table 1, EvoPlan covers five out of the seven considered features. In particular, our approach is able to *infer migration*, make use of *incremental plan* by considering the *popularity* and *issues*, so as to eventually recommend an *upgrade plan*. Compared to the existing tools, EvoPlan tackles most of the issues previously presented.

### 3 Proposed approach

In this paper we propose an approach to support the first phase of the migration process, i.e., inferring the possible upgrade plans that can satisfy the request of the developer that might want to upgrade a given TPL used in the project under development.

Our approach aims at suggesting the most appropriate migration plan by taking into consideration two key factors: the *popularity* of the upgrade plan and the

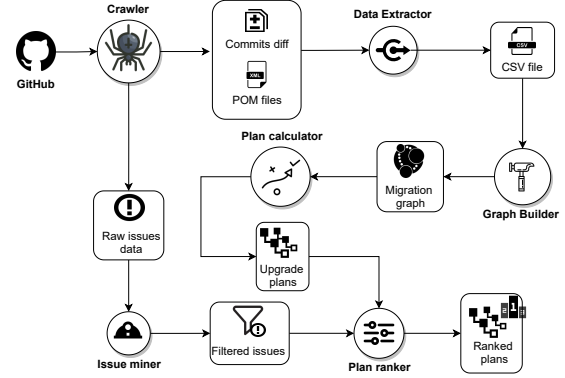


Fig. 2 EvoPlan’s architecture.

*availability of discussions* about it. Popularity means how many clients have performed a given upgrade plan, while discussions are GitHub issues that have been open and closed in projects during the migration phase. By mining GitHub using the dedicated API,<sup>11</sup> we are able to extract the information required as input for the recommendation engine of EvoPlan.

The conceived approach is depicted in Fig. 2 and consists of six components, i.e., *Crawler*, *Data Extractor*, *Graph Builder*, *Issues Miner*, *Plan Calculator* and *Plan Ranker*. With the *Crawler* component, the system retrieves information about GitHub repositories and downloads them locally. These repositories are then analyzed by the *Data Extractor* component to excerpt information about commits and history version. Once all the required information has been collected, *Graph Builder* constructs a migration graph with multiple weights, and *Issues Miner* generates data related to GitHub issues. The *Plan Calculator* component relies on the graph to calculate the k-best paths available. Finally, *Plan Ranker* sorts these paths by considering the number of issues. In the succeeding subsections, we are going to explain in detail the functionalities of each component.

#### 3.1 Crawler

Migration-related information is mined from GitHub using the *Crawler* component. By means of the JGit library,<sup>12</sup> *Crawler* downloads a set  $P$  of GitHub projects that have at least one *pom.xml* file, which is a project file containing the list of all adopted TPLs. In case there are multiple *pom.xml* files, they will be analyzed separately to avoid information loss. Then, the *Crawler* component analyzes all the repository’s commits that

<sup>11</sup> <https://developer.github.com/v3/>

<sup>12</sup> <https://www.eclipse.org/jgit/>

affect the *pom.xml* to find added and removed TPLs. Additionally, raw issue data is obtained and stored in separate files. In particular, we count the number of opened and closed issues for each project  $p \in P$  in a specific time interval  $D$ . The starting point of this interval is identified when a certain version  $v$  of a given library  $l$  that is added as dependencies of the *pom.xml* file in client  $C$ . A previous study [12] demonstrates that the monthly rate of open issues tends to decrease over time. Thus, the endpoint of  $D$  is obtained by considering the first two months of development to extract relevant data concerning the considered library  $l$  without loss of data. In such a way, the GitHub issues that have been opened and closed for each TLP that has been added in  $p$ , are obtained for further processing phases.

### 3.2 Data Extractor

In this phase, data is gathered by means of *JGit*, and analyzed using different processing steps as follows. The first step makes use of the GitHub *log* command to retrieve the list of every modification saved on GitHub for a specific file. Furthermore, the command provides the code *SHA* for every commit, which allows us to identify it. For instance, Fig. 3.a depicts a commit related to a given *pom.xml* file taken as input. The identifier of the commit is used to retrieve the list of the corresponding operated changes as shown in Fig. 3.b. In particular, inside a commit we can find a large number of useful information like what was written or removed and when. The *Data Extractor* component focuses on the lines which contain an evidence of library changes. In a commit, the added lines are marked with the sign '+', whereas the removed ones are marked with '-' (see the green and red lines, respectively shown in Fig. 3.b). In this way, the evolution of a library is obtained by analyzing the sequence of added/removed lines. With this information, *EvoPlan* is also able to count how many clients have performed a specific migration. The information retrieved by the *Data Extractor* component is stored in a target CSV file, which is taken as input by the subsequent entity of the process as discussed below.

### 3.3 Graph Builder

This component creates nodes and relationships by considering the date and library changes identified in the previous phase. To this end, *EvoPlan* exploits the Cypher query language<sup>13</sup> to store data into a NEO4J graph. For

<sup>13</sup> <https://neo4j.com/developer/cypher-query-language/>

```
rick@rick-HP-EliteBook-830-G5:~/juddi$ git log pom.xml
commit a28db0a7a6f30a15bcd588057a1cfb410a9ff28
Merge: 2c8d0f433 444c35a72
Author: spyhunter99 <spyhunter99@users.noreply.github.com>
Date: Sun Mar 15 16:48:15 2020 -0400

Merge pull request #7 from JLLeitschuh/fix/JLL/use_https_to_resolve_dependencies

[SECURITY] Use HTTPS to resolve dependencies in Maven Build
```

a) Example of *log*

```
rick@rick-HP-EliteBook-830-G5:~/juddi$ git diff a28db0a7a6f30a15bcd588057a1cfb410a9ff28
diff --git a/docs/asciidoc/ClientGuide/pom.xml b/docs/asciidoc/ClientGuide/pom.xml
index dd09f6d82..193da5bf5 100644
--- a/docs/asciidoc/ClientGuide/pom.xml
+++ b/docs/asciidoc/ClientGuide/pom.xml
@@ -19,7 +19,7 @@
 <parent>
   <groupId>org.apache.juddi.juddi-docs</groupId>
   <artifactId>juddi-guide-parent</artifactId>
-  <version>3.0.0</version>
+  <version>3.0.0</version>
 </parent>
 <artifactId>juddi-client-guide</artifactId>
 <packaging>jdocbook</packaging>
```

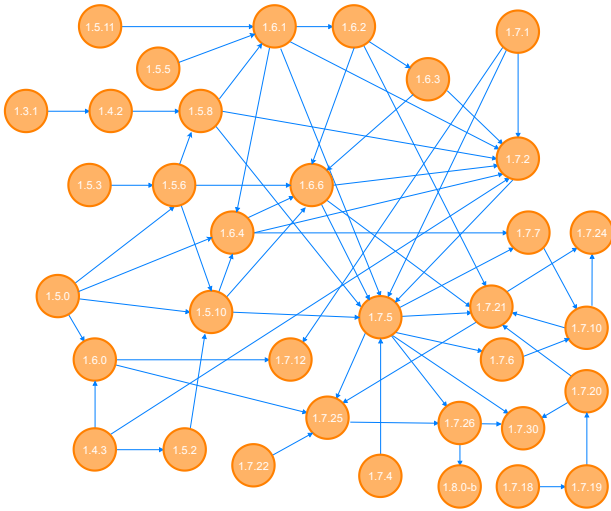
b) Example of *diff*

**Fig. 3** Example of artifacts used by the *Data Extractor* component.

instance, we extract from CSV files two pairs library-version  $(l, v1)$  and  $(l, v2)$  with signs '-' and '+', respectively. In this way, the component creates an oriented edge from  $(l, v1)$  to  $(l, v2)$ . Once the first edge is created, any further pairs containing the same library upgrade will be added as an incremented weight on the graph edge. The date value contained in the CSV record is used to avoid duplicated edges or loops. Furthermore, each edge is weighted according to the number of clients as described in *Data Extractor* phase. That means if we find  $w$  times the same couple  $(l, v1)$  to  $(l, v2)$  (i.e., a number of  $w$  projects have already migrated the library  $l$  from  $v1$  to  $v2$ ), the edge will have a weight of  $w$ . Thus, the final outcome of this component is a migration graph that considers the community's interests as the only weight. For instance, Fig. 4 represents the extracted migration graph for the *slf4j-api* library. The graph contains all the mined version of the library and for each pair the corresponding number of clients that have performed the considered upgrade is shown. For instance, in Fig. 4 the edge from the version *1.6.1* to *1.6.4* is selected, and 14 clients (see the details on the bottom) have performed such a migration.

### 3.4 Plan Calculator

Such a component plays a key role in the project. Given a library to be upgraded, the starting version, and the target one, *Plan Calculator* retrieves the  $k$  shortest paths by using the well-founded *Yen's K-shortest paths algorithm* [31] which has been embedded into the NEO4J library. As a default heuristic implemented in *EvoPlan*, the component retrieves all the possible paths that maximize the popularity of the steps that can be performed to do the wanted upgrade. Thus, the *Plan Calculator* component employs the aforementioned weights which



**Fig. 4** Migration graph of the *slf4j* library.

represent the popularity as a criteria for the shortest path algorithm.

By considering the graph shown in Fig. 4, there are several possibilities to upgrade *slf4j* from version *1.5.8* to *1.7.25*. By taking into account the available weights, EvoPlan can recommend the ranked list depicted in Fig. 5. The first path in the list suggests to follow the steps *1.6.1*, *1.6.4*, and *1.7.5* to reach the final version considered in the example, i.e., *1.7.25*.<sup>14</sup> Such a plan is the one that is performed most by other projects, which rely on *slf4j* and that have already operated the wanted library migration. Thus, such a path is more frequent than directly updating the library to the newest version.

### 3.5 Issues Miner

Issues play an important role in project development. For instance, by solving issues, developers contribute to the identification of bugs as well as the enhancement of software quality through feature requests [16]. In the scope of this work, we exploit issues as criteria for ordering upgrade plans. In particular, we rely on the availability of issues that have been opened and closed due to upgrades of given third-party libraries.

The *Issue Miner* component is built to aggregate and filter raw issues data gathered in the early stage of the process shown in Fig. 2. However, due to the internal construction of NEO4J, we cannot directly embed this data as a weight on the migration graph’s edges.

<sup>14</sup> It is worth noting that the popularity values are disproportionate to the popularity of the corresponding upgrade plans. In the example shown in Fig. 5 the most popular upgrade is the one with popularity value 0.898.

Table 2 Issues information extracted for *commons-io*.

| Version | Open Issues | Closed Issues | Delta |
|---------|-------------|---------------|-------|
| 1.0     | 14          | 33            | 19    |
| 1.3.2   | 150         | 420           | 270   |
| 1.4     | 87          | 408           | 321   |
| 2.0     | 5           | 10            | 5     |
| 2.0.1   | 133         | 457           | 324   |
| 2.1     | 129         | 516           | 387   |
| 2.2     | 67          | 999           | 932   |
| 2.3     | 5           | 20            | 15    |
| 2.4     | 939         | 3,283         | 2,344 |
| 2.5     | 64          | 918           | 854   |
| 2.6     | 64          | 548           | 484   |

Thus, as shown in Section 3.1, we collect the number of open and closed issues considering a specific time window, i.e., two months starting from the introduction of a certain TLP in the project. Then, this component filters and aggregates the issues data related by using Pandas, a widely-used Python library for data mining [21]. For instance, Table 2 shows the mined issues related to the *commons-io* library. In particular, for each version of the library, the number of issues that have been opened and closed by all the analysed clients since they have migrated to that library version is shown. EvoPlan can employ the extracted data to enable a ranking function based on GitHub issues as discussed in the next section.

*Issues Miner* works as a stand-alone component, thus it does not impact on the time required by the overall process. In this way, we have an additional source of information that can be used later in the process as a supplementary criterion to choose the ultimate upgrade plan from the ranked list produced by the *Plan Calculator* component.

### 3.6 Plan Ranker

In the final phase, the k-paths produced by the *Plan Calculator* are rearranged according to the information about issues. For every path, we count the average value of opened/closed issues. A large value means that a certain path potentially requires less integration effort since there are more closed issues than the opened ones [16], i.e., issues have been tackled/solved rather than being left untouched.

Thus, the aim is to order the plans produced by *Plan Calculator* according to the retrieved issues: among the most popular plans we will propose those with the highest issue values.

Table 3 shows an example of the ranking process. There are two highlighted paths, the gray row corresponds to the best result according to the plan popularity only. In fact, the gray highlighted plan is the one with lower popularity value. Meanwhile, the orange row



Fig. 5 List of  $k$ -shortest paths for *slf4j*.

Table 3 An example of the ranking results.

| Proposed Path                             | Pop. Value | Issues Value |
|---|------------|--------------|
| 1.5.8, 1.6.1, 1.6.4, 1.6.6, 1.7.5, 1.7.25 | 1.446      | 57           |
| 1.5.8, 1.6.1, 1.6.4, 1.7.5, 1.7.25        | 0.898      | 58           |
| 1.5.8, 1.7.5, 1.7.25                      | 1.0        | 58           |
| 1.5.8, 1.6.1, 1.7.5, 1.7.25               | 1.0        | 61           |
| 1.5.8, 1.6.1, 1.6.4, 1.7.2, 1.7.5, 1.7.25 | 1.238      | 58           |

is recommended according to the issues criteria (in this case, the higher the issue value, the better). The path that should be selected is the orange one because it represents the one characterized by the highest activity in terms of opened and closed issue, among the most popular ones. In this way, EvoPlan is able to recommend an upgrade plan to migrate from the initial version to the desired one by learning from the experience of other projects which have already performed similar migrations.

## 4 Evaluation

To the best of our knowledge, there are no replication packages and reusable tools related to the approaches outlined in Section 2 that we could use to compare EvoPlan with them. As a result, it is not possible to compare EvoPlan with any baselines. Thus, we have to conduct an evaluation of the proposed approach on a real dataset collected from GitHub. Section 4.1 presents three research questions, while Section 4.2 describes the evaluation process. Section 4.3 gives a detailed description of the dataset used for the evaluation, and the employed metrics are specified in Section 4.4.

### 4.1 Research questions

To study the performance of EvoPlan, we consider the following research questions:

- **RQ<sub>1</sub>**: *How effective is EvoPlan in terms of prediction accuracy?* To answer this question, we conduct experiments following the ten-fold cross-validation methodology [13] on a dataset considering real migration data collected from GitHub. Moreover, we compute *Precision*, *Recall*, and *F-measure* by comparing the recommendation outcomes with real migrations as stored in GitHub;
- **RQ<sub>2</sub>**: *Is there any correlation between the GitHub issues and the popularity of a certain migration path?* We analyze how the number of opened and closed issues could affect the migration process. To this end, we compute three different statistical coefficients to detect if there exists any correlation among the available data.
- **RQ<sub>3</sub>**: *Is EvoPlan able to provide consistent recommendations in reasonable time?* Besides the recommended migration steps, we are interested in measuring the time of the overall process, including the graph building phase. This aims at ascertaining the feasibility of our approach in practice.

### 4.2 Overall process

As depicted in Fig. 6, we perform experiments using the ten-fold cross-validation methodology on a well-founded dataset coming from an existing work [14]. Given the whole list of  $\approx 11,000$  projects, we download the entire dataset using the *Crawler* component. Then, the dataset is split into testing and ground truth projects, i.e., 10% and 90% of the entire set, respectively, by each round of the process. This means that

in each round we generate a new migration graph by using the actual 90% portion. Given a single testing project, the *Analyzing commits* phase is conducted to capture the actual upgrade path followed by the repository, as stated in Section 3.1. To build the ground-truth graph, i.e., the real migration in GitHub, we consider projects not included in the testing ones and calculate every possible upgrade plan for each TPLs.

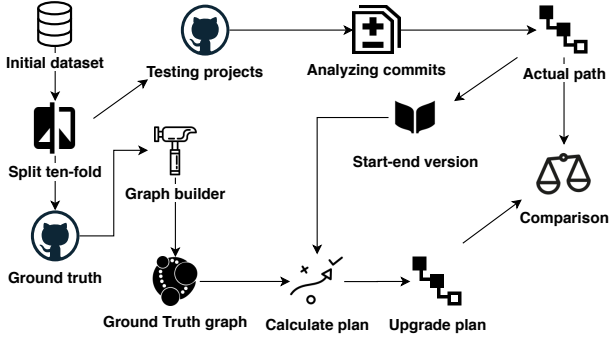


Fig. 6 The evaluation process.

To aim for a reliable evaluation, we select the starting and the end version of a certain TPL from the actual plan of a testing project. The pair is used to feed the *Plan Calculator* component which in turn retrieves the proposed plan. In this respect, by following the two paths we are able to compute the metrics to assess the overall performance, namely precision, recall, and F-measure.

#### 4.3 Data collection

We make use of an existing dataset which has been curated by a recent study available on GitHub.<sup>15</sup> The rationale behind this selection is the quality of the repositories which were collected by applying different filters, i.e., removing duplicates, including projects with at least one *pom.xml* file, and crawling only well-maintained and mature projects. Table 4 summarizes the number of projects and *pom.xml* files. The dataset consists of 10,952 GitHub repositories, nevertheless we were able to download only 9,517 of them, as some have been deleted or moved. Starting from these projects, we got a total number of 27,129 *pom.xml* files. Among them, we selected only those that did not induce the creation of empty elements by the *Data Extractor* component while analyzing *logs* and *diffs* as shown in Fig. 3. The filtering process resulted in 13,204 *pom.xml* files. The training

set is used to create a migration graph to avoid any possible bias. For each round, we tested 420 projects, and 3,821 projects are used to build the graph.

Table 4 Statistics of the dataset.

|   |        |
|---|--------|
| Total number of projects                | 10,952 |
| Number of downloaded projects           | 9,517  |
| Total number of <i>pom.xml</i> files    | 27,129 |
| Number of screened <i>pom.xml</i> files | 13,204 |

Table 5 summarizes the set of libraries in the dataset, obtained by employing the *Crawler* module (cf. Section 3.1). There are seven popular libraries,<sup>16</sup> i.e., *junit*, *httpclient*, *slf4j*, *log4j*, *commons-io*, *guava*, and *commons-lang3*. Among others, *junit* has the largest number of migrations, i.e., 2,972. Concerning the number of versions, *slf4j* has 71 different versions, being the densest library. Meanwhile, *commons-lang3* is associated with the smallest number of migrations, i.e., 162, and *commons-io* is the sparsest library with only 16 versions. The last column shows the number of versions that we could exploit to get the issues. The difference means that no issues data was available for the whole versions dataset.

Table 5 Number of migrations and versions.

| Library              | # migrations | # versions | # issue vers. |
|----------------------|--------------|------------|---------------|
| <i>junit</i>         | 2,972        | 30         | 19            |
| <i>httpclient</i>    | 218          | 53         | 35            |
| <i>slf4j</i>         | 209          | 71         | 26            |
| <i>log4j</i>         | 229          | 42         | 19            |
| <i>commons-io</i>    | 186          | 16         | 11            |
| <i>guava</i>         | 627          | 70         | 34            |
| <i>commons-lang3</i> | 162          | 16         | 13            |

#### 4.4 Metrics

Given a migration path retrieved by EvoPlan, we compare it with the real migration path extracted from a testing project. To this end, we employ *Precision*, *Recall*, and *F-measure* (or *F<sub>1</sub>-score*) widely used in the Information Retrieval domain to assess the performance prediction of a system. In the first place, we rely on the following definitions:

<sup>15</sup> <https://bit.ly/20pd1GH>

<sup>16</sup> <https://mvnrepository.com/popular>

- A *true positive* corresponds to the case when the recommended path matches with the actual path extracted from the testing projects; *TP* is the total number of true positives;
- A *false positive* means that the recommended upgrade plan is not present in the ground-truth paths; *FP* is the total number of false positives;
- A *false negative* is the migration steps that should be present in the suggested plan but they are not; *FN* is the total number of false negatives.

Considering such definitions, the aforementioned metrics are computed as follows:

$$P = \frac{TP}{TP + FP} \quad (1)$$

$$R = \frac{TP}{TP + FN} \quad (2)$$

$$F - measure = \frac{2 \times P \times R}{P + R} \quad (3)$$

**Rank correlation:** We consider the following coefficients:

- *Kendall's tau* measures the strength of dependence between two variables. It is a non-parametric test, i.e., it is based on either being distribution-free or having a specified distribution but with the distribution's parameters unspecified.
- *Pearson's correlation* is the most widely used correlation statistic to measure the degree of the relationship between linearly related variables. In particular, this coefficient is suitable when it is possible to draw a regression line between the points of the available data.
- *Spearman's correlation* is a non-parametric test that is used to measure the degree of association between two variables. Differently from Pearson's coefficient, Spearman's correlation index performs better in cases of monotonic relationships.

All the considered coefficients assume values in the range  $[-1, +1]$ , i.e., from perfect negative correlation to perfect positive correlation. The value 0 indicates that between two variables there is no correlation.

In the next section, we explain in detail the experimental results obtained through the evaluation.

## 5 Experimental results

We report and analyze the obtained results by answering the research questions introduced in the previous section.

### 5.1 **RQ<sub>1</sub>**: How effective is EvoPlan in terms of prediction accuracy?

Table 6 reports the average results obtained from the cross-validation evaluation. EvoPlan achieves the maximum precision for *commons-io*, i.e., 0.90 in all the rounds. The tool also gets a high precision for *junit*, i.e., 0.88. Meanwhile, the smallest precision, i.e., 0.58 is seen by *httpclient*. Concerning recall, EvoPlan obtains a value of 0.94 and 0.96 for the *junit* and *commons-io* libraries, respectively. In contrast, the tool achieves the worst recall value with *httpclient*, i.e., 0.64. Overall by considering the F-Measure score, we see that EvoPlan gets the best and the worst performance by *commons-io* and *httpclient*, respectively.

**Table 6** Precision, Recall, and F-Measure considering popularity.

| Library              | Precision   | Recall      | F-measure   |
|----------------------|-------------|-------------|-------------|
| <i>junit</i>         | 0.88        | 0.94        | 0.91        |
| <i>httpclient</i>    | 0.58        | 0.64        | 0.61        |
| <i>slf4j-api</i>     | 0.65        | 0.74        | 0.69        |
| <i>log4j</i>         | 0.88        | 0.93        | 0.91        |
| <i>commons-io</i>    | <b>0.90</b> | <b>0.96</b> | <b>0.94</b> |
| <i>guava</i>         | 0.60        | 0.73        | 0.65        |
| <i>commons-lang3</i> | 0.66        | 0.67        | 0.65        |

Altogether, we see that there is a substantial difference between the performance obtained by EvoPlan for different libraries. We suppose that this happens due to the availability of the training data. In particular, by carefully investigating each library used in the evaluation, we see that the libraries with the worst results in terms of performance have a few migrations that we can extract from the *pom.xml* on average (cf. Table 5). For instance, there are 162 and 209 migrations associated with *commons-lang3* and *slf4j-api*, respectively and EvoPlan gets a low performance on these libraries. Meanwhile, there are 2,972 migrations for *junit* and EvoPlan gets high precision, recall, and  $F_1$  for this library. It means that less data can negatively affect the final recommendations.

Another factor that can influence the conducted evaluation could be the number of versions involved in an upgrade for each library i.e., the availability of fewer versions dramatically reduce the migration-related information. This hypothesis is confirmed by the observed values for *log4j* and *junit* that bring better results with 39 and 40 analyzed versions respectively. However, there is an exception with *guava*, i.e., EvoPlan yields a mediocre result for the library ( $F_1=0.65$ ), though we considered 627 migration paths and 49 different versions. By examining the library, we realized that it has many versions

employed in the Android domain as well as abandoned versions. Thus, we attribute the reduction in performance to the lack of decent training data.

**Answer to RQ<sub>1</sub>.** EvoPlan is capable of predicting the correct upgrade plan given a real-world migration dataset. Although for some libraries we witness a reduction in the overall performances, the main reason can be found in the lack of migration paths in the original dataset.

### 5.2 RQ<sub>2</sub>: Is there any correlation between the GitHub issues and the popularity of a certain migration path?

To answer this question we measure the correlation among observed data, i.e., the number of clients that perform a certain migration step and the issues delta considering the time interval described in Section 3.1. The number of clients performing migration is defined with the term *popularity* as described in Section 3.4. Meanwhile, as its name suggests, the *delta* is the difference between the number of closed issues and the number of open ones. It assumes a positive value when the number of closed issues is greater than the opened ones. In contrast, negative values are observed when open issues exceed the number of closed ones. In other words, deltas characterizes migration steps in terms of closed issues.

The results of the three indexes are shown in Table 7. As we can see, all the metrics show a positive correlation between the number of clients that perform a certain migration and the corresponding delta issues. In particular, Kendall’s tau  $\tau$  is equal to 0.458, while Spearman’s rank  $\rho$  reaches the value of 0.616. The maximum correlation is seen by Pearson’s coefficient, i.e.,  $r = 0.707$ .

The strong correlation suggests that given a library, the more clients perform a migration on its versions, the more issues are solved. As it has been shown in a recent work [16], the act of solving issues allows developers to identify bugs and improve code, as well as enhance software quality. Summing up, having a large number of migrated clients can be interpreted as a sign of maturity, i.e., the evolution among different versions attracts attention by developers.

**Table 7** Correlation coefficients with a  $p$ -value  $< 2.2e-16$ .

| Metric              | Value |
|---------------------|-------|
| Kendal’s ( $\tau$ ) | 0.458 |
| Pearson ( $r$ )     | 0.707 |
| Spearman ( $\rho$ ) | 0.616 |

**Answer to RQ<sub>2</sub>.** There is a significant correlation between the upgrade plan popularity and the number of closed issues. This implies that plans to be given highest priority should be those that have the majority of issues solved during the migration.

### 5.3 RQ<sub>3</sub>: Is EvoPlan able to provide consistent recommendations in reasonable time?

We measured the average time required for running experiments using a mainstream laptop with the following information: i5-8250U, 1.60GHz Processor, 16GB RAM, and Ubuntu 18.04 as the operating system. Table 8 summarizes the time for executing the corresponding phases.

**Table 8** Execution time.

| Phase          | Time (seconds) |
|----------------|----------------|
| Graph building | 15,120         |
| Querying       | 0.11           |
| Testing        | 145.44         |

The most time consuming phase is the creation of graph with 15,120 seconds, corresponding to 252 minutes. Meanwhile, the querying phase takes just 0.11 seconds to finish; the testing phase is a bit longer: 145.44 seconds. It is worth noting that the testing consists of the sub-operations that are performed in actual use, i.e., opening CSV files, extracting the actual plan, and calculating the shortest path. This means that we can get an upgrade plan in less than a second, which is acceptable considering the computational capability of the used laptop. This suggests that EvoPlan can be deployed in practice to suggest upgrade plans.

**Answer to RQ<sub>3</sub>.** The creation of the migration graph is computationally expensive. However, it can be done offline, one time for the whole cycle. EvoPlan is able to deliver a single upgrade plan in a reasonable time window, making it usable in the field.

### 5.4 Threats to validity

This section discusses possible threats that may affect the proposed approach. Threats to *internal validity* could come from the graph building process. In particular, the crawler can retrieve inaccurate information from *pom.xml* files or GitHub commits. To deal with this, we employed a similar mining technique used in some related studies presented in Section 2.2, i.e., Meditor, APIwave, aiming to minimize missing data. Another possible pitfall lies in downgrade migrations, i.e.,

a client that moves from a newer version to an older one. We consider the issue as our future work.

Concerning *external validity*, the main threat is related to the generalizability of the obtained results. We try to mitigate the threat by considering only popular Java libraries. Nevertheless, EvoPlan relies on a flexible architecture that can be easily modified to incorporate more TPLs. Concerning the employed GitHub issues data, they are coarse-grained, i.e., we can have a huge number of issues that do not have a strong tie with the examined TLPs. We addressed this issue in the paper by considering the ratio of the delta instead of absolute numbers. Concerning the supported data sources, EvoPlan employs *Maven* and GitHub to mine migration histories and retrieve issues, respectively. Thus, currently, upgrade plans can be recommended for projects that rely on these two technologies. However, the architecture of EvoPlan has been designed in a way that supporting additional data sources would mean operating localized extensions in the *Crawler*, *Data Extractor*, and *Issue Miner* components without compromising the validity of the whole architecture.

Finally, threats to *construct validity* concern the ten-fold cross-validation shown in Section 4.2. Even though this technique is used mostly in the machine learning domain, we mitigate any possible incorrect values by considering a different ground-truth graph for each evaluation round. Additionally, the usage of GitHub issues could be seen as a possible threat. We mitigate this aspect by using such information as post-processing to reduce possible negative impacts on the recommended items, i.e., ranking the retrieved upgrade plans according to the total amount of issues.

## 6 Related work

A plethora of studies highlights different issues related to the TLPs migration problem. Dig and Johnson [10] demonstrate the role of code refactorings as the principal origin of *breaking changes*, i.e., failures caused by a library upgrade from an older version to a newer one. *Binary incompatibilities (BIs)* happen when the application is no longer compilable after migration [5]. The Clirr tool has been used to detect the entities that cause incompatibilities by analyzing the JAR files of the testing project. By evaluating six different recommendation techniques that are typically used to fix BIs, this study exhibits that they were capable of resolving only 20% of them.

A recent work [14] attracts the community attention over the *migration awareness* problem. By conducting a user study, the two main migration awareness mechanisms have been evaluated, i.e., security advisories and

new releases announcement. In this respect, the results show that the majority of the software systems rarely update the older but reliable libraries and security advisories provide incomplete solutions to the developers.

Alrubaye *et al.* [1] conducted an empirical study to highlight the benefits of the migration process over software quality measured by the three standard metrics used in the domain, i.e., coupling, cohesion, and complexity. By relying on a dataset composed of nine different libraries and 57,447 Java projects, statistical tests have been carried on relevant migration data. The results confirm that the migration process improves the code quality in terms of the mentioned metrics.

The problem of *Technical debt* has been studied in both academia and industry [3], and it is related to “immature” code sent to production [6]. Although this approach is used to achieve immediate results, it could lead to future issues after a certain period. To solve this, technical debt can be repaid through code refactorings by carrying out a cost-benefit analysis. Lavazza *et al.* [15] propose the usage of technical debt as an external software quality attribute of a project. Furthermore, technical debt can affect software evolution and maintainability by introducing defects that are difficult to fix.

Sawant and Bacchelli [25] investigate *API features usages* over different TLPs releases by mining 20,263 projects and collect 1,482,726 method invocations belonging to five different libraries. Using the proposed tool fine-GRAPE, two case studies have been conducted considering two aspects, i.e., the number of migrations towards newer versions and the usages of API features. The results confirm that developers tend not to update their libraries. More interesting, the second study shows that a low percentage of API features are actually used in the examined projects.

## 7 Conclusion and future work

The migration of TPLs during the development of a software project plays an important role in the whole development cycle. Even though some tools are already in place to solve the issue, different challenges are still opened, i.e., reducing the effort during the migration steps or the need to consider heterogeneous data sources to name a few. We proposed EvoPlan, a novel approach to support the upgrading of TPLs by considering miscellaneous software artifacts. By envisioning different components, our tool is capable of extracting relevant migration data and encoding it in a flexible graph-based representation. Such a migration graph is used to retrieve multiple upgrade plans considering the popularity as the main rationale. They are eventually ranked by

exploiting the GitHub issues data to possibly minimize the effort that is required by the developer to select one of the candidate upgrade plans. A feasibility study shows that the results are promising, with respect to both effectiveness and efficiency.

As future work, we plan to incorporate additional concepts in the migration graph, i.e., TLPs documentation, Stack Overflow posts, and issues sentiment analysis. We believe that such additional data allows EvoPlan to better capture the migration paths performed by clients. Moreover, we can consider a larger testing dataset to improve the coverage of the recommendation items, i.e., provide upgrade plans for more TLPs.

**Acknowledgements** The research described in this paper has been partially supported by the AIDOaRT Project, which has received funding from the European Union's H2020-ECSEL-2020, Federal Ministry of Education, Science and Research, Grant Agreement n°101007350

## References

1. Alrubaye, H., Alshoaibi, D., Alomar, E., Mkaouer, M.W., Ouni, A.: How does library migration impact software quality and comprehension? an empirical study. In: S. Ben Sassi, S. Ducasse, H. Mili (eds.) *Reuse in Emerging Software Engineering Practices*, pp. 245–260. Springer International Publishing, Cham (2020)
2. Alrubaye, H., Mkaouer, M.W., Khokhlov, I., Reznik, L., Ouni, A., McGoff, J.: Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing* **90**, 106–140 (2020)
3. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* **6**(4), 110–138 (2016). DOI 10.4230/DagRep.6.4.110
4. Collie, B., Ginsbach, P., Woodruff, J., Rajan, A., O’Boyle, M.F.: M3: Semantic api migrations. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 90–102 (2020)
5. Cossette, B.E., Walker, R.J.: Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In: *Procs. of the ACM SIGSOFT 20th Int. Symposium on the Foundations of Software Engineering - FSE ’12*, p. 1. Cary, North Carolina (2012). DOI 10.1145/2393596.2393661
6. Cunningham, W.: The wycash portfolio management system. *SIGPLAN OOPS Mess.* **4**(2), 29–30 (1992). DOI 10.1145/157710.157715. URL <https://doi-org.univaq.cla.s.cineca.it/10.1145/157710.157715>
7. Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P., Rubei, R.: TopFilter: An Approach to Recommend Relevant GitHub Topics. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM ’20*. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3382494.3410690
8. Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P.T., Rubei, R.: Development of recommendation systems for software engineering: the CROSSMINER experience **26**(4), 69. DOI 10.1007/s10664-021-09963-7. URL <https://doi.org/10.1007/s10664-021-09963-7>
9. Di Sipio, C., Rubei, R., Di Ruscio, D., Nguyen, P.T.: Using a Multinomial Naïve Bayesian (MNB) Network to Automatically Recommend Topics for GitHub Repositories. In: *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering, EASE2020, Trondheim, Norway, April 15-17, 2020, EASE’20*, pp. 24–34. ACM (2020). DOI 10.1145/3383219.3383227
10. Dig, D., Johnson, R.: The role of refactorings in API evolution. In: *21st IEEE Int. Conf. on Software Maintenance (ICSM’05)*, pp. 389–398 (2005). DOI 10.1109/ICSM.2005.90
11. Hora, A., Valente, M.T.: Apiwave: Keeping track of API popularity and migration. In: *2015 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, pp. 321–323 (2015). DOI 10.1109/ICSM.2015.7332478
12. Kikas, R., Dumas, M., Pfahl, D.: Issue dynamics in github projects. In: *Proceedings of the 16th International Conference on Product-Focused Software Process Improvement - Volume 9459, PROFES 2015*, p. 295–310. Springer-Verlag, Berlin, Heidelberg (2015). DOI 10.1007/978-3-319-26844-6\_22. URL [https://doi.org/10.1007/978-3-319-26844-6\\_22](https://doi.org/10.1007/978-3-319-26844-6_22)
13. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI’95*, p. 1137–1143. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995)
14. Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* **23**(1), 384–417 (2018). DOI 10.1007/s10664-017-9521-5
15. Lavazza, L., Morasca, S., Tosi, D.: Technical debt as an external software attribute. In: *Proceedings of the 2018 International Conference on Technical Debt - TechDebt ’18*, pp. 21–30. ACM Press, Gothenburg, Sweden (2018). DOI 10.1145/3194164.3194168. URL <http://dl.acm.org/citation.cfm?doid=3194164.3194168>
16. Liao, Z., He, D., Chen, Z., Fan, X., Zhang, Y., Liu, S.: Exploring the Characteristics of Issue-Related Behaviors in GitHub Using Visualization Techniques. *IEEE Access* **6**, 24003–24015 (2018). DOI 10.1109/ACCESS.2018.2810295. Conference Name: IEEE Access
17. Misra, V., Reddy, J.S.K., Chimalakonda, S.: Is there a correlation between code comments and issues?: an exploratory study. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 110–117. ACM, Brno Czech Republic (2020). DOI 10.1145/3341105.3374009. URL <https://dl.acm.org/doi/10.1145/3341105.3374009>
18. Neil, L., Mittal, S., Joshi, A.: Mining Threat Intelligence about Open-Source Projects and Libraries from Code Repository Issues and Bug Reports. In: *2018 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pp. 7–12 (2018). DOI 10.1109/ISI.2018.8587375
19. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Di Penta, M.: CrossRec: Supporting Software Developers by Recommending Third-party Libraries. *Journal of Systems and Software* p. 110460 (2019). DOI <https://doi.org/10.1016/j.jss.2019.110460>. URL <http://www.sciencedirect.com/science/article/pii/S0164121219302341>
20. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., Di Penta, M.: FOCUS: A recommender system for mining API function calls and usage patterns. In:

- Proceedings of the 41st international conference on software engineering, ICSE '19, pp. 1050–1060. IEEE Press, Piscataway, NJ, USA (2019)
21. Pandas: pandas documentation — pandas 1.1.3 documentation (2020). URL <https://pandas.pydata.org/docs/>
  22. Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M.: Prompter: Turning the IDE into a self-confident programming assistant. *Empirical Software Engineering* **21**(5), 2190–2231 (2016). DOI 10.1007/s10664-015-9397-1. URL <http://link.springer.com/10.1007/s10664-015-9397-1>
  23. Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T. (eds.): *Recommendation Systems in Software Engineering*. Berlin, Heidelberg (2014). DOI 10.1007/978-3-642-45135-5
  24. Rubei, R., Di Sipio, C., Nguyen, P.T., Di Rocco, J., Di Ruscio, D.: PostFinder: Mining Stack Overflow posts to support software developers. *Information and Software Technology* **127**, 106367 (2020). DOI <https://doi.org/10.1016/j.infsof.2020.106367>. URL <http://www.sciencedirect.com/science/article/pii/S0950584920301361>
  25. Sawant, A.A., Bacchelli, A.: fine-GRAPE: fine-grained API usage extractor – an approach and dataset to investigate API usage. *Empirical Software Engineering* **22**(3), 1348–1371 (2017). DOI 10.1007/s10664-016-9444-6. URL <https://doi.org/10.1007/s10664-016-9444-6>
  26. Teyton, C., Falleri, J.R., Blanc, X.: Mining Library Migration Graphs. In: 2012 19th Working Conf. on Reverse Engineering, pp. 289–298 (2012). DOI 10.1109/WCRE.2012.38
  27. Xavier, L., Brito, A., Hora, A., Valente, M.T.: Historical and impact analysis of api breaking changes: A large-scale study. In: 2017 IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER), pp. 138–147 (2017)
  28. Xing, Z., Stroulia, E.: Umldiff: An algorithm for object-oriented design differencing. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, p. 54–65. Association for Computing Machinery, New York, NY, USA (2005). DOI 10.1145/1101908.1101919. URL <https://doi.org/10.1145/1101908.1101919>
  29. Xing, Z., Stroulia, E.: API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering* **33**(12), 818–836 (2007). DOI 10.1109/TSE.2007.70747
  30. Xu, S., Dong, Z., Meng, N.: Meditor: Inference and Application of API Migration Edits. In: 2019 IEEE/ACM 27th Int. Conf. on Program Comprehension (ICPC), pp. 335–346 (2019). DOI 10.1109/ICPC.2019.00052
  31. Yen, J.Y., YENt, J.Y.: Finding the k shortest loopless paths in a network (2007)