Information and Software Technology 177 (2025) 107588

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

DeepMig: A transformer-based approach to support coupled library and code migrations

Juri Di Rocco [a], Phuong T. Nguyen [a], Claudio Di Sipio [a], Riccardo Rubei [a], Davide Di Ruscio [a,*], Massimiliano Di Penta [b]

[a] *Università degli studi dell'Aquila, 67100 L'Aquila, Italy*
[b] *Università degli studi del Sannio, Italy*

ARTICLE INFO

Keywords:
Recommender system
Library migration
Transformers

ABSTRACT

Context: While working on software projects, developers often replace third-party libraries (TPLs) with different ones offering similar functionalities. However, choosing a suitable TPL to migrate to is a complex task. As TPLs provide developers with Application Programming Interfaces (APIs) to allow for the invocation of their functionalities after adopting a new TPL, projects need to be migrated by the methods containing the affected API calls. Altogether, the coupled migration of TPLs and code is a strenuous process, requiring massive development effort. Most of the existing approaches either deal with library or API call migration but usually fail to solve both problems coherently simultaneously.
Objective: This paper presents DeepMig, a novel approach to the coupled migration of TPLs and API calls. We aim to support developers in managing their projects, at the library and API level, allowing them to increase their productivity.
Methods: DeepMig is based on a transformer architecture, accepts a set of libraries to predict a new set of libraries. Then, it looks for the changed API calls and recommends a migration plan for the affected methods. We evaluate DeepMig using datasets of Java projects collected from the Maven Central Repository, ensuring an assessment based on real-world dependency configurations.
Results: Our evaluation reveals promising outcomes: DeepMig recommends both libraries and code; by several projects, it retrieves a perfect match for the recommended items, obtaining an accuracy of 1.0. Moreover, being fed with proper training data, DeepMig provides comparable code migration steps of a static API migrator, a baseline for the code migration task.
Conclusion: We conclude that DeepMig is capable of recommending both TPL and API migration, providing developers with a practical tool to migrate the entire project.

1. Introduction

During the life-cycle of software projects, developers usually need to replace third-party libraries (TPLs) with other ones providing similar functionalities. This happens due to various reasons. Among others, old TPLs might no longer be maintained, have compatibility issues, or are longer suitable for the project [1], or need to be changed because of licensing compatibility problems [2,3]. Indeed, choosing which TPLs to migrate is challenging because an inappropriate replacement would cause incompatibility or breaking changes, among other possible problems. Substituting a library with a new one while retaining the same behavior is called *library migration* [4–7], which is considered a daunting task [8].

Studies have shown that developers are reluctant to migrate TPLs [1,9], not only due to the fear of incompatibility and breaking changes [10,11] but also because migration requires extra effort and responsibility. As a result, developers usually resort to being in their *comfort zone*, retaining the most familiar versions and neglecting the increasing *maintenance debt* [12]. Recent studies [1,13] showed that 81.5% of the surveyed systems remain with popular older versions, and more than half of the projects never migrate more than 50% their TPLs. Procrastinating updates of TPLs may culminate in ripple effects, harming software in various aspects [1,14].

When TPLs are migrated, it is also necessary to operate changes to the affected source code that might no longer work, triggering the

* Corresponding author.
E-mail addresses: juri.dirocco@univaq.it (J. Di Rocco), phuong.nguyen@univaq.it (P.T. Nguyen), claudio.disipio@univaq.it (C. Di Sipio), riccardo.rubei@univaq.it (R. Rubei), davide.diruscio@univaq.it (D. Di Ruscio), dipenta@unisannio.it (M. Di Penta).

https://doi.org/10.1016/j.infsof.2024.107588
Received 14 February 2024; Received in revised form 17 September 2024; Accepted 20 September 2024
Available online 28 September 2024

Table 1
Library migration.

C_1	ID	C_2	ID	Action
commons-lang:2.6	1	commons-lang:2.6	1	No change
junit:4.12	8	junit:4.12	8	No change
jackson-databind:2.8.1	9	jackson-databind:2.8.1	9	No change
jackson-core:2.8.1	12	jackson-core:2.8.1	12	No change
unirest-java:1.4.9	5	okhttp:3.9.1	37	*Migrated*
(log4j)		slf4j	11	*Newly added*
(null)		commons-codec:1.10	11	*Newly added*

need for *API migration* [15]. Thus, there are two levels of migration to deal with: *(i)* the *library level*; and *(ii)* the *source code level*. For the former, developers need to replace a library with a more suitable one. For the latter, developers need to adapt the client's affected source code to conform with the new libraries and the related APIs.

Several approaches have been proposed to recommend TPL migrations [1,5–7], and they learn migrations by mining the history of similar projects. Thus, the recommended libraries come from projects with a similar set of TPLs. Nevertheless, this does not necessarily guarantee that developers can concurrently perform migration at the API call level: even projects using the same TPLs may invoke the APIs differently. Altogether, the migration of TPLs and the affected source code is a challenging task [16–18], making a mechanism to automate the process as a whole highly desirable.

We propose DeepMig as a holistic approach to migrating software projects. DeepMig is based on a *Transformer* [19] deep learning architecture, that has been conceived for machine translation [20]. DeepMig is a dual-purpose tool that provides suitable recommendations to migrate *both TPLs and API calls*. We built a double-layer architecture by mining projects' development history and changes at the method level. Once trained, the networks predict the set of TPLs to be included in the project and API calls for the affected code. DeepMig outperforms a well-founded TPL migration baseline, and it provides similar code migration plans compared to a static API migrator. To our best knowledge, DeepMig is the first tool to recommend migrations at both library and source code levels.

This paper makes the following contributions:

– A novel approach to provide developers with a complete solution to third-party library and API call migrations, built on top of the transformer architecture.
– A tailored process to mine library and API migration data from Maven Central Repository.[1]
– An empirical evaluation and comparison with state-of-the-art baselines using real-world datasets.
– The DeepMig tool and the curated datasets have been published online to allow future research [21].

2. Motivation and background

This section introduces some basic concepts (Section 2.1), and a motivating example (Section 2.2) as a base for further presentation.

2.1. Terminology

We briefly recall the most basic concepts in library and API migration by referring to the code snippet in Fig. 1(a);

- An **API** is a code unit that can be used following a defined interface without knowing its implementation [22,23]. Each API consists of public methods M available to clients, e.g., the `asString()` method of the `HttpResponse` type (Line 34);
- A **method definition**[2] is made of a name, a list of parameter types, a return type, and a body. A method may call others, e.g., the `sendRequest()` method (Line 1) invokes `getHTTPMethod()` within its body (Line 17);
- A **method declaration** is the signature of a defined method, containing a name, a list of parameter types, and a return type;
- A **third-party library** or *TPL* is an encapsulated software module offering functionalities to reuse;
- An **API method invocation** or *invocation* is a call made from a definition $d \in D$ to a method $m \in M$. An *API field access* is an entry to a field $f \in F$ from a method declaration.

2.2. Motivating example

Let us consider a developer who is working on the RECOMBEE/JAVA-API-CLIENT project,[3] and let us assume that at the time of investigation, $\mathbf{C}_1$ (`com.recombee:api-client:1.4.0`) is the client that the developer needs to upgrade. As shown in Table 1, the developer replaces the TPL `unirest-java:1.4.9` with `okhttp:3.9.1` and adds `commons-codec:1.10` to develop new functionalities. Because of such changes, $\mathbf{C}_1$ has to be migrated to fix the code affected by the TPL replacement, leading to $\mathbf{C}_2$ (`com.recombee:api-client:2.0.0`)–the future version of $\mathbf{C}_1$. The developer has to conduct two levels of migration as follows.

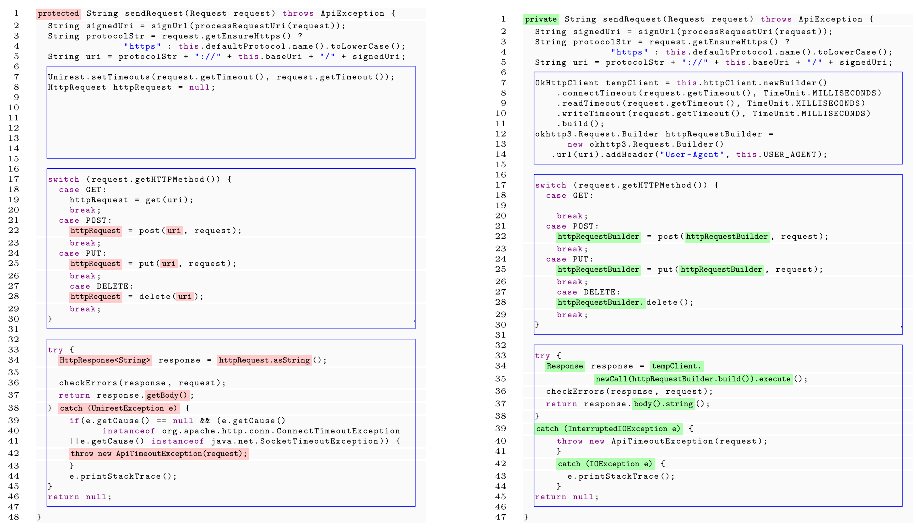
▷ **Library migration**. Since `unirest-java:1.4.9`[4] is outdated, the developer plans to replace the library. This is an uphill task, as the developer has to investigate different libraries and understand if they have the desired functionalities, implying an enduring process [5,24].

Table 1 depicts the list of libraries of the old and new clients, i.e., before and after the migration. Compared to the old one, by the new client $\mathbf{C}_2$, while the first four libraries remain unchanged, `unirest-java:1.4.9` is removed, and the new library `okhttp:3.9.1`[5] is added. By carefully inspecting the two libraries, we see that they implement similar features, i.e., establishing, sustaining, and terminating connections via the *HTTP* protocol. Essentially, there is a migration from `unirest-java:1.4.9` to `okhttp:3.9.1`, aiming to provide the project with similar but more updated/stable functionalities.

Apart from `okhttp:3.9.1`, `commons-codec:1.10` is another newly added library. This is understandable in light of the following reason: `unirest-java:1.4.9` includes functionalities that are offered by `commons-codec:1.10`, while `okhttp:3.9.1` does not. Thus, removing `unirest-java:1.4.9` means there must be a new library added, i.e., `commons-codec:1.10`, which can compensate for the missing code. This scenario represents a straightforward one-to-many library migration. In the context of third-party library migration, we distinguish the following types of library mappings:

[1] https://mvnrepository.com/repos/central

[2] The terms "*method definition*", "*method*", and "*definition*" are used interchangeably across the paper for the sake of presentation.
[3] https://github.com/recombee/java-api-client/
[4] http://kong.github.io/unirest-java/
[5] https://square.github.io/okhttp/



```java
protected String sendRequest(Request request) throws ApiException {
  String signedUri = signUrl(processRequestUri(request));
  String protocolStr = request.getEnsureHttps() ?
                "https" : this.defaultProtocol.name().toLowerCase();
  String uri = protocolStr + "://" + this.baseUri + "/" + signedUri;

  Unirest.setTimeouts(request.getTimeout(), request.getTimeout());
  HttpRequest httpRequest = null;








  switch (request.getHTTPMethod()) {
    case GET:
      httpRequest = get(uri);
      break;
    case POST:
      httpRequest = post(uri, request);
      break;
    case PUT:
      httpRequest = put(uri, request);
      break;
      case DELETE:
      httpRequest = delete(uri);
      break;
  }


  try {
    HttpResponse<String> response = httpRequest.asString();

    checkErrors(response, request);
    return response.getBody();
  } catch (UnirestException e) {
      if(e.getCause() == null && (e.getCause()
            instanceof org.apache.http.conn.ConnectTimeoutException
      ||e.getCause() instanceof java.net.SocketTimeoutException)) {
      throw new ApiTimeoutException(request);
      }
      e.printStackTrace();
  }
  return null;

}
```

(a) Original definition (`com.recombee:api-client:1.4.0`)

```java
private String sendRequest(Request request) throws ApiException {
  String signedUri = signUrl(processRequestUri(request));
  String protocolStr = request.getEnsureHttps() ?
                "https" : this.defaultProtocol.name().toLowerCase();
  String uri = protocolStr + "://" + this.baseUri + "/" + signedUri;

  OkHttpClient tempClient = this.httpClient.newBuilder()
      .connectTimeout(request.getTimeout(), TimeUnit.MILLISECONDS)
      .readTimeout(request.getTimeout(), TimeUnit.MILLISECONDS)
      .writeTimeout(request.getTimeout(), TimeUnit.MILLISECONDS)
      .build();
  okhttp3.Request.Builder httpRequestBuilder =
        new okhttp3.Request.Builder()
     .url(uri).addHeader("User-Agent", this.USER_AGENT);


  switch (request.getHTTPMethod()) {
    case GET:

      break;
    case POST:
      httpRequestBuilder = post(httpRequestBuilder, request);
      break;
    case PUT:
      httpRequestBuilder = put(httpRequestBuilder, request);
      break;
      case DELETE:
      httpRequestBuilder.delete();
      break;
  }


  try {
    Response response = tempClient.
                newCall(httpRequestBuilder.build()).execute();
    checkErrors(response, request);
    return response.body().string();
  }
  catch (InterruptedIOException e) {
      throw new ApiTimeoutException(request);
      }
      catch (IOException e) {
        e.printStackTrace();
      }
  return null;

}
```

(b) Migrated definition (`com.recombee:api-client:2.0.0`)

Fig. 1. Code migration for the `sendRequest()` method definition of the RECOMBEE/JAVA-API-CLIENT project.

– *one-to-one mapping*: This migration directly replaces one third-party library in the original system with a corresponding library in the new version of the client. This occurs when there is a direct equivalent that performs the same functionality, facilitating a simple and direct transition.
– *one-to-many mapping*: This type of mapping replaces a single third-party library with multiple libraries in the new system. This approach is adopted when the replaced library's functionalities are broader and can be better served by several specialized libraries in the new system, enhancing specific aspects of the original library's capabilities.
– *many-to-one mapping*: This mapping consolidates several third-party libraries from the original setup into a single comprehensive library in the new system. It is applied when the new library offers an integrated suite of functions that can effectively replace and simplify the functionalities provided by multiple older libraries.
– many-to-many mapping: This scenario aims to replace multiple third-party libraries with an equal or greater number of new ones in the new system. This mapping is necessary when the transition involves significant realignment of the provided functionalities, such as splitting, combining, or extending features across several new libraries, to adapt to the system's technological advancements or architectural changes.

It is worth mentioning that *library migration* [6] is different from *library upgrade* [8]. In particular, with the latter, developers refine the system by upgrading some of the current TPLs with their newest versions. Instead, with the former, developers replace some of the used TPLs with alternative ones.

▷ **Code migration**. Migrating TPLs is the first step of the considered upgrading, which also requires changes at the code level. For instance, by replacing `unirest-java:1.4.9` with `okhttp:3.9.1`, the developer needs to inspect the source code of $\mathbf{C}_1$ and looks for affected definitions, and conducts adaptations.

The client $\mathbf{C}_1$ has many affected method definitions. We consider, as an example (shown in Fig. 1(a)), the `sendRequest()` definition needs to be migrated as it invokes APIs of the removed `unirest-java:1.4.9` library.

Similarly to library migration challenges, in the context of API function migration, Alrubaye et al. [25] classified the migration of API function calls scenarios as follows:

– *one-to-one mapping*: This mapping involves directly replacing a single API function call from the old library with a corresponding function from the new library. This is the simplest form of mapping, where each function in the original library has a direct equivalent in the replacement library.
– *one-to-many mapping*: In this scenario, a single API function call to the removed library is replaced by multiple API function calls to the new library. These additional calls may include functions from newly added libraries or existing APIs (e.g., standard Java API calls). This type of mapping is used when the new library offers more granular functionality split across several functions.
– *many-to-one mapping*: This type of mapping occurs when multiple API functions from the original library are consolidated into a single function in the new library. This may happen when the new library provides a more integrated or comprehensive function that can accomplish what previously required several separate functions.
– *many-to-many mapping*: This complex mapping occurs when multiple API functions from the original library are replaced by an equal or greater number of API functions in the new library. Such mappings are typically necessary when the functionalities of the old library are either split into more discrete functions or combined and extended in the new library and other ones (including Java API), requiring comprehensive changes in how the functions interact and operate. This ensures that the application can leverage the full capabilities of the new library while maintaining or enhancing its original functionality.

Fig. 1(b) shows the changes occurring in the commit `a0576aa`,[6] referring to the final code after migration. In particular, the access modifier of `sendRequest()` is changed from `protected` to `private`, and the APIs related to `unirest-java:1.4.9` are also adapted to those offered by `okhttp:3.9.1`.[7]

In Fig. 1(a) and Fig. 1(b), pairwise blue frames are used to mark the portions of code being adapted. By comparing the upper frames, we can see that all the declarations related to `unirest-java:1.4.9`

[6] https://bit.ly/3OaVv2e

[7] To facilitate the reading, we employ a GitHub-like color scheme to highlight the changes between two versions of the same method definition, i.e., red and green correspond to removed and added code, respectively.

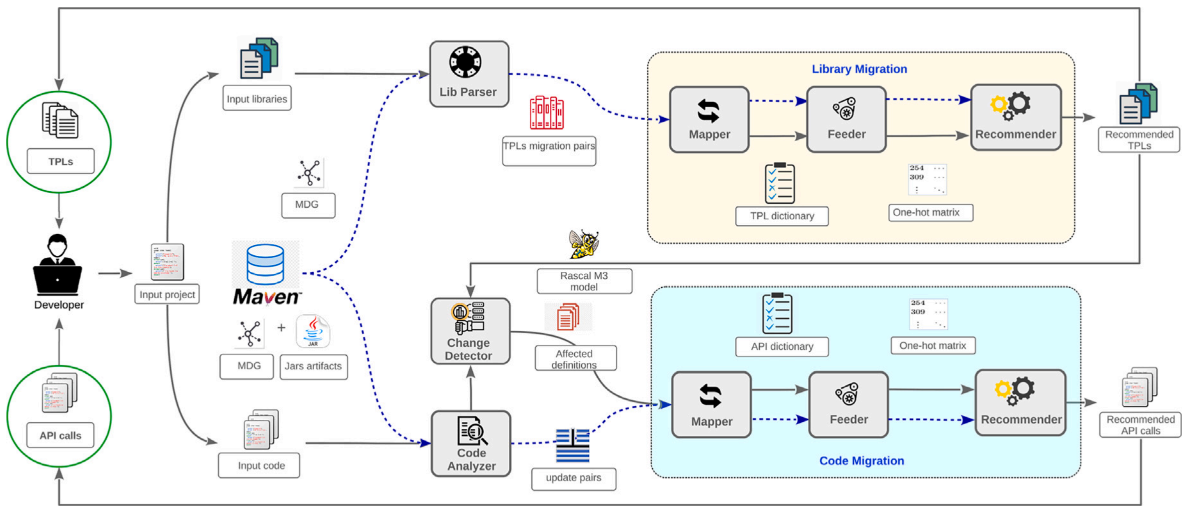


Fig. 2. The DeepMig double-layer architecture to provide library and API migration.

have been replaced with those from `okhttp:3.9.1` (Lines 6–15). By the middle frames, the old variables are superseded, i.e., the `httpRequestBuilder` variable of the apiokhttp3.Request.Builder type substitutes for the `HttpRequest` variable of the `HttpRequest` type (Lines 22–28). Finally, at the bottom, the affected API calls are migrated accordingly, e.g., `getBody()` is changed to `body().string()` (Line 37).

The `sendRequest()` method is only one among the affected definitions, and adapting all of them is an arduous task. Despite refactoring services and tools exist, e.g., OpenRewrite[8] can be used to support code migration, their recommendations are based on a set of prescribed rules. Therefore, while offering a practical solution to migration, such services have their own *Achilles heel*, i.e., they will not work if there are not sufficiently defined rules prepared beforehand.

Remark 1. The migration of libraries and the affected code is a daunting task, as it requires intensive manual work, to search for suitable libraries, and update the related code. Thus, approaches aimed at assisting developers to automatically perform the migration are greatly needed.

There is a common characteristic between the two types of migration as follows. Library migration can be considered as a mapping from a set of libraries to the new ones, e.g., changing from `unirest-java:1.4.9` to `okhttp:3.9.1`, and from `null` to `commons-codec:1.10`. This is also the case with code migration, e.g., `getBody()` is transformed to `body().string()`, and `HttpRequest` is superseded by apiokhttp3.Request.Builder.

Remark 2. **(i)** The migration of TPLs boils down to transforming an input to an output sequence of libraries; ***(ii)*** Likewise, code migration is equal to the transition from a sequence of API invocations to a different one.

The motivating example inspires us to automate the migration of both levels together. We come up with a holistic solution to library and code migration, exploiting machine translation techniques, which are briefly recalled as follows.

In this paper, we postulate that the problem of library and API call migration can be reformulated as a machine translation task. We develop DeepMig following a holistic approach, applying a transformer architecture to recommend migration for libraries and the affected source code.

3. Proposed approach

DeepMig is a double-layer system consisting of different modules to mine the development history of projects as well as changes in the method level to provide relevant recommendations (see Fig. 2). In the learning phase (the blue dashed lines), training data related to library and code migrations is collected from open-source repositories, e.g., Maven to feed as input for the whole system. In the deployment phase (the continuous lines), data from developers is converted into a suitable format, which is then used as input for the recommendation engine.

3.1. Lib Parser

To learn relationships between libraries we leverage data from the Maven Central Repository. In particular, we use the Maven Dependency Graph (MDG) developed by [26], an open-source dataset that stores Maven artifacts in a graph database. The MDG includes all types of relations between Maven artifacts, i.e., upgrades and dependencies. Also, the MDG can *(i)* link each version of a specific library to the next version (if it exists); and *(ii)* join an artifact to their dependencies.

The Lib Parser component of DeepMig uses the MDG to mine the migration history of TPLs, by considering the following pieces of information:

- *Artifact*: Each software stored in MDG is called *client* if it plays the role of a final product, or *library* if it is a dependency of another artifact.
- *Version*: An instance of an artifact issued at a certain time.
- *An update pair* $\langle c_x, c_y \rangle$ is two consecutive versions x and y of an artifact c.
- $ML_{\langle c_x,c_y \rangle} = AL_{\langle c_x,c_y \rangle} \cup RL_{\langle c_x,c_y \rangle} \cup UL_{\langle c_x,c_y \rangle}$: the set of TPLs that has been either added (AL) in c_y, or removed (RL) from c_x, or upgraded (UL) in an update pair $\langle c_x, c_y \rangle$.

With respect to the dependencies listed in Table 1, $RL_{\langle C_1,C_2 \rangle}$ consists of `unirest-java:1.4.9`, $AL_{\langle C_1,C_2 \rangle}$ includes `okhttp:3.9.1` and `commons-codec:1.10`, while $UL_{\langle C_1,C_2 \rangle}$ is empty. Lib Parser mines MDG to find update pairs $\langle c_x, c_y \rangle$ where $ML_{\langle c_x,c_y \rangle} \neq \emptyset$. It can even take 2 libraries as input, i.e., l_r and l_a to restrict the set of update pairs by selecting ones where $l_r \in RL_{\langle c_x,c_y \rangle}$ and $l_a \in AL_{\langle c_x,c_y \rangle}$. In particular, DeepMig takes a list of libraries and predicts a suggested list of replacements. This capability allows DeepMig to support more complex

[8] https://docs.openrewrite.org/

Table 2
Example of dictionaries generated by the Mapper.

	Library	Frequency	ID
$\mathcal{D}_L$	org.slf4j:slf4j-api:1.7.25	6,179	0
	commons-lang:commons-lang:2.6	**5,924**	**1**
	...	...	...
	junit:junit:4.12	**3,271**	**8**
	...	...	...
	com.sun.jersey:jersey-grizzly2:1.11	5	35 542
	API call	Frequency	ID
$\mathcal{D}_A$	java/lang/StringBuilder/toString()	28,827	0
	java/lang/StringBuilder/append(java.lang.String)	28,690	1
	...	...	...
	http/exceptions/UnirestException/printStackTrace()	6	60 725
	beaker/server/Proposer/configuration()	5	60 735

scenarios, including *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many* library migrations. It is worth noting that DeepMig does not identify which new libraries replace the existing ones, i.e., it does not compute the matches between them. Instead, its scope is to provide developers with insights for upgrading the libraries involved in the current version of the project under development.

3.2. Code analyzer

Starting from source code, DeepMig extracts related information including definitions, API invocations, and field accesses (see Section 2.1). The DeepMig Code Analyzer relies on the Rascal M^3 model [27] to mine data from open-source repositories. M^3 manages the `declaration`$_R$ and `methodInvocation`$_R$ relationship containing a set of pairs $\langle loc_1, loc_2 \rangle$, i.e., loc_1 and loc_2 represent locations, which are uniform resource identifiers (URI) to specify locations of definitions and invocations. A `declaration`$_R$ relation maps the location of a method to its URI. Likewise, a `methodInvocation`$_R$ relation associates the artifact identities of a *caller* with those of the *callee(s)*. This enables Code Analyzer to retrieve the referred resources without contextual information.

3.3. Change detector

We decided to implement a tailored change detector, rather than relying on pre-existing code differencing tools, such as GumTree [28]. The rationale behind such the decision is as follows: We are interested in extracting API function calls of method definitions provided by a migrated library, not textual syntax that includes parameter values, i.e., those that GumTree can handle. In this respect, our change detector can identify two consecutive versions of the same method definition that uses the APIs provided by a migrated library. Given an update pair $\langle c_x, c_y \rangle$ the Code Migration Pairs (*CMP*) are defined as follows:

- For each $l \in RL_{\langle c_x, c_y \rangle}$, the migration pair *MP* includes $\langle MD_{c_x}, MD_{c_y} \rangle$, where the MD_{c_x} definition calls an API of l, and MD_{c_y} holds in c_y with the same signature. Code Analyzer does not consider definitions that have been removed to support the migration, as they do not contribute to the training models to adapt existing method definitions.
- For each $l \in AL_{\langle c_x, c_y \rangle}$, the migration pairs *MP* includes $\langle MD_{c_x}, MD_{c_y} \rangle$, pairs where the method definition MD_{c_y} calls an API of l and the method definition MD_{c_x} holds in c_x with the same signature. The Code Analyzer does not consider method definitions that have been added to support the migration. In fact, they do not contribute to the training models to adapt existing method definitions.
- For each $l \in UL_{\langle c_x, c_y \rangle}$, *MP* encompasses the $\langle MD_{c_x}, MD_{c_y} \rangle$ pair, where the method definitions MD_{c_y} and MD_{c_x} have the same signature and call an API of l. For instance, the two implementations of `sendRequest()` depicted in Fig. 1 are identified as a migration pair since c_1 uses the APIs provided by the removed library `unirest-hjava:1.4.9`, and `sendRequest()` is still available in c_2. Moreover, c_2 calls the APIs provided by `okhttp:3.9.1` the new library. It is important to remark that Change Detector does not consider pairs where the method definition is unchanged in c_x and c_y as they do not contribute to the training models to adapt existing definitions. Finally, Change Detector extracts the list of migration pairs that shows how the evolved version of a method definition deals with the migration of the library. The generated migration pair is provided as input for Mapper, which is described in the succeeding subsection.

3.4. Artifact mapper

An artifact[9] is usually a long string, and thus it is not feasible to feed it directly as input to Transformer. Therefore, we devise a method to convert the original names to a shorter ones using numbers. In particular, each artifact has to be encoded using a unique number, thus converting each input sequence into a chain of numbers. To optimize the computation, we need to keep the input as short as possible, i.e., the length of the input sentence should be small. In fact, many artifacts are very popular, and they appear several times in source code. As a result, if we use large numbers (i.e., with many digits) to encode them, then we would encounter long sequence, necessitating more resources to handle the data. Given a training corpus, two dictionaries are built, i.e., $\mathcal{D}_L$ for libraries, and $\mathcal{D}_A$ for API calls. Mapper counts the number of occurrences for each artifact and builds a list of artifacts with their frequency, sorted in descending order. From the list, the Artifact Mapper assigns an ID to each artifact, by traversing down from the top to bottom. The IDs start from 0 and increase by 1 at every artifact. In this way, a more frequent API call is encoded with a smaller number, and vice versa. Altogether, this aims to avoid long sequences, thereby optimizing the computation. The two dictionaries are independent on each other, i.e., they are used separately, and they can encode artifacts sharing the same numbers, without interfering with each other.

An example of $\mathcal{D}_L$ and $\mathcal{D}_A$ is shown in Table 2, where artifacts that appear in the motivating example are highlighted in bold face, to facilitate the presentation in the next subsection. The `slf4j-api:1.7.25` library is the most frequent one as it appears 6,179 times, thus being encoded with 0, `commons-lang:2.6` is encoded with ID=1; Meanwhile, `jersey-grizzly2:1.11` has ID=35542, as it counts only 5 times across the corpus. Likewise for APIs, `java/lang/StringBuilder/toString()` is the most common API call, thus ID=0; while `beaker/server/Proposer/configuration()` is very rare, and it is given the ID=60735.[10]

[9] For the sake of presentation, in the rest of this paper, we call a library or an API the common name *artifact*, unless otherwise stated.

[10] For the sake of clarity and presentation, we do not present all the related IDs. The complete list of TPLs and their corresponding IDs is available in the online appendix.

$$\overbrace{\boxed{1}\boxed{\text{␣}}\boxed{8}\boxed{\text{␣}}\boxed{9}\boxed{\text{␣}}\boxed{12}\boxed{\text{␣}}\boxed{5}}^{\text{Input sequence}}\boxed{\Rightarrow}\overbrace{\boxed{1}\boxed{\text{␣}}\boxed{8}\boxed{\text{␣}}\boxed{9}\boxed{\text{␣}}\boxed{12}\boxed{\text{␣}}\boxed{37}\boxed{\text{␣}}\boxed{11}}^{\text{Output sequence}}$$

Fig. 3. Input and output sequences for the example in Table 1.

$$\overbrace{\boxed{20}\boxed{\text{␣}}\boxed{39}\boxed{\text{␣}}\boxed{133}\boxed{\text{␣}}\boxed{402}\boxed{\text{␣}}\boxed{88}}^{\text{Input sequence}}\boxed{\Rightarrow}\overbrace{\boxed{20}\boxed{\text{␣}}\boxed{39}\boxed{\text{␣}}\boxed{155}\boxed{\text{␣}}\boxed{\ldots}\boxed{\text{␣}}\boxed{324}\boxed{\text{␣}}\boxed{34}}^{\text{Output sequence}}$$

Fig. 4. Input and output sequences for the example in Fig. 1.

3.5. Attention and transformer

One of the main challenges in machine translation [20] is to capture the relationship among words within sentences to learn their co-occurrence properly. To cope with this problem, the attention model was conceived as an effective means to associate the mutually relevant parts. For instance, in the following sentence: "*The dog does not play with the ball, because **it** feels bored*", the word "*it*" refers to "*the dog*" instead of "*the ball*". Attention can enforce this relationship, and Transformer [19] goes one step further, being made of multiple blocks of encoders/decoders with self-attention modules to obtain promising performance in NLP. The decoder has an extra attention block, focusing on the relevant part of the sequence. The embedded words of the input are fed to the encoder and propagated to all the encoders.

Further details about the transformer architecture can be found in the original paper introducing it [19].

Thanks to its features, Transformer has the potential to capture the relationship among libraries or API calls when they are migrated. Both types of migration are related to time series data, i.e., the transition over the course of time. Thus, giving proper attention to some entries in a sequence allows us to catch their collocation, e.g., "`okhttp:3.9.1` *and* `commons-codec:1.10` *are migrated from* `unirest-java:1.4.9`" can be conveniently learnt with attention and transformer.

3.6. Feeder

This component converts an input list of artifacts (TPLs, API calls) to machine-processable format including *(i)* sequences; and *(ii)* tensors. First, by looking up the two dictionaries $\mathcal{D}_L$ and $\mathcal{D}_A$, the Feeder translates artifacts into a sequence of numbers. Fig. 3 shows the input and output sequences of the motivating example of Table 1. All the IDs are retrieved from $\mathcal{D}_L$, i.e., the dictionary for libraries. Migrating the TPLs of $\mathbf{C}_1$ to $\mathbf{C}_2$ now corresponds to translating the input sequence X="*1␣8␣9␣12␣5*" to the output sequence Y="*1␣8␣9␣12␣37␣11*", where "␣" represents a space to distinguish between the IDs inside a sequence.

Similarly, the sequences for the code in Fig. 1 are generated by looking up $\mathcal{D}_A$, as shown in Fig. 4, i.e., the input sequence corresponds to the original definition (Fig. 1(a)), while the output sequence represents the migrated definition (Fig. 1(b)).[11]

A transformer consists of several long short-term memory (LSTM) units [29], and all the IDs need to be converted to one-hot vectors with 0 and 1 as follows. First, we traverse through all the sequences to compute Θ, the length of the longest sequence. Afterward, C_v, the corpus of all the characters used to form the IDs, is built to contain 11 letters for encoding ten numbers, i.e., from 0 to 9, and the white space, i.e., $C_v = \{0, 1, \ldots, 9, \text{␣}\}$. Then, each character is converted to a one-hot vector whose length corresponds to the corpus's number of characters, i.e., $\Pi = |C_v|$=11. Finally, a sequence is represented as a 2D matrix of size $\Theta \times \Pi$, where each row corresponds to the one-hot vector of a character within a sequence, and the columns represent the vocabularies. Sequences, whose length is smaller than Θ, are padded with "␣" to fill the gap.

$$\begin{array}{cc|ccccccccccc|}
\text{IDs} & \text{Vector} & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} & \mathbf{7} & \mathbf{8} & \mathbf{9} & \text{␣} \\
\mathbf{1} & \mathrm{x}_1 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{␣} & \mathrm{x}_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
\mathbf{8} & \mathrm{x}_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\
\text{␣} & \mathrm{x}_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
\mathbf{9} & \mathrm{x}_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\
\text{␣} & \mathrm{x}_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
\mathbf{1} & \mathrm{x}_7 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathbf{2} & \mathrm{x}_8 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{␣} & \mathrm{x}_9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
\mathbf{5} & \mathrm{x}_{10} & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\
\text{␣} & \mathrm{x}_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
\text{␣} & \mathrm{x}_{12} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1}
\end{array}$$

Fig. 5. One-hot matrix for X="*1␣8␣9␣12␣5␣␣*".

Fig. 5 depicts the matrix representation of X="*1␣8␣9␣12␣5*", i.e., the input sequence in Fig. 3. Let Θ be 12,[12] then since $len(X) = 10 < \Theta$, the last two rows x_{11} and x_{12} are filled with "␣".

Finally, given a training corpus of N sequences, Feeder renders all of them into a suitable format as in Fig. 6. There, each slide corresponding to a 2D matrix is used to represent data for a sequence, and all the slides form a tensor of size $\Theta \times \Pi \times N$, that is then provided as input for the recommendation engine. The data model in Fig. 6 can be used to represent the input for both for library and API call migration to feed DeepMig.

3.7. Recommender

This component is built on top of Transformer [19], which contains blocks of encoders/decoders as shown in Fig. 7. Both the Encoder and Decoder are chains of LSTM units, each of which is made of self-attention layers and feed-forward neural networks. The input data is first fed to a self-attention layer, to better memorize a word by looking at other related words. Fig. 7 illustrates the Transformer-based engine [20] to translate the input API sequence to the output one for the example in Fig. 3. Essentially, the series of the Encoder is used to transform the input data into numbers, which are then used by the Decoders to generate the output sequence. DeepMig is suitable for analyzing a set of existing API function calls and predicting appropriate replacements. This capability empowers DeepMig to support a wide

[11] These sequences are produced similarly to the one for library migration.

[12] Note that this is a small value of Θ used for illustration purposes only. In practice, the sequence length Θ is usually chosen as a large number enough to capture most of the real-world sequences.

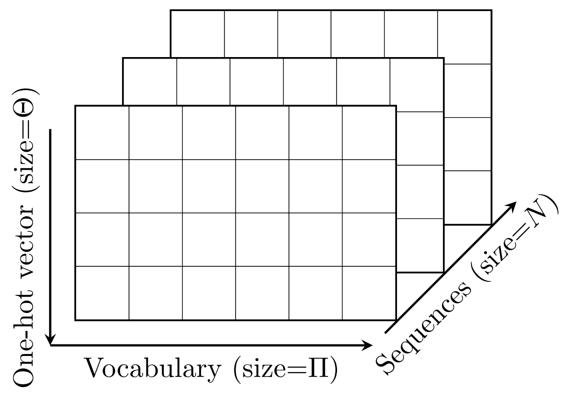


Fig. 6. Tensor to store data for a complete training set.

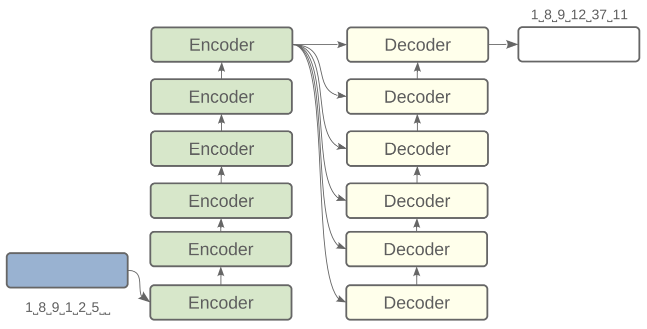


Fig. 7. The Transformer-based recommendation engine.

array of complex API migration scenarios, such as*one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many* mappings. Analogously to the library migration recommender, it is important to clarify that DeepMig does not explicitly determine which new API function calls replace the existing ones; it does not directly link new API calls to specific functionalities of the old APIs. Instead, its core functionality is to provide valuable insights for updating the API function calls across the entire project. In other words, DeepMig assists developers in systematically upgrading and modernizing their software applications without proving API function call mappings.

4. Evaluation

This section describes the empirical study aimed at evaluating DeepMig.

4.1. Research questions

To evaluate DeepMig, we aim to answer the following research questions.

- **RQ$_1$**: *How well does DeepMig perform in migrating libraries?* First, through literature analysis, we study to what extent the issue of TPL migration has ever been addressed by state-of-the-art research. The ultimate goal is to find a suitable comparison baseline for DeepMig, for what concerns library migration. Second, we evaluate the ability of DeepMig to migrate libraries using a real-world dataset collected from Maven.
- **RQ$_2$**: *When can DeepMig generate API migration steps comparable to those recommended by OpenRewrite?* The recommendations by OpenRewrite can be considered as reference migrations, and we study under which circumstances DeepMig can produce similar results obtained with OpenRewrite. This is meaningful in practice as automatic migration helps users avoid the efforts of developing static refactoring rules.
- **RQ$_3$**: *What are the applicability and limitations of using ChatGPT for API migration?* We study the feasibility of using ChatGPT to support the migration of API function calls. While a direct comparison of DeepMig with OpenRewrite and ChatGPT is not possible due to the different nature of the three approaches, we assess the generated migration in terms of accuracy.
- **RQ$_4$**: *How can DeepMig improve its performance?* In code recommendation, it is important to suggest API calls relevant to the given development scenario. We evaluate in which context DeepMig can enhance its performance, providing developers with more useful code to complete their tasks.

4.2. Baselines for comparison

To the best of our knowledge, no existing approach is able to recommend both library and code migrations. As a result, we have to perform the comparison singularly, i.e., evaluating the two types of migrations using different baselines. This section explains the process we followed to look for suitable tools to be directly compared with DeepMig.

4.2.1. Library migration

After a preliminary search, we did not find any approach to deal with the problem of library migration, which can be used as a benchmark for evaluating DeepMig. Thus, we decided to perform a literature analysis to understand the extent to which the issue of library migration has been addressed by state-of-the-art research. Though we did not target a complete, detailed systematic literature review, we followed existing guidelines for such type of studies in Software Engineering research [30–32], to cover a wide range of existing work. For the sake of a reasonable trade-off between efficiency and coverage of state-of-the-art studies on library migration, we employed a search strategy led by four W-questions [33], i.e., "*Which?*" "*Where?*" "*What?*" "*When?*" explained as follows.

- *Which?* We performed a comprehensive search using a combination of automated and manual methods to gather research papers from a variety of sources, including conferences and journals.
- *Where?* The literature analysis was performed on premier software engineering venues including *(i)* nine conferences: ASE, ESEC/FSE, ESEM, ICSE, ICSME, ICST, ISSTA, MSR, and SANER; and *(ii)* six journals: EMSE, ESWA, IST, JSS, TOSEM, and TSE. All the considered venues are detailed in Table 3. The *Scopus* database[13] was chosen for the automatic search, and all the papers published by a given year of a given venue were retrieved through the advanced Scopus search and export features. An excerpt of the queries is shown in Listing 1.
- *What?* Title and abstract of each article were fetched following a set of predefined keywords.
- *When?* Since automated library migration is a recent research theme, the search was confined to the most seven recent years, i.e., from 2017 to 2023.

Starting from the publication venues, we counted 19,834 articles for the six most recent years, i.e., from 2017 to 2023. We then restricted our attention to research contributions related only to TPL migration. Thus, from the collected corpus we narrowed down the scope by using four sets of keywords as follows: *(i)* **GOAL**: "*migrat**", "*version**, or "*evolve*"; *(ii)* **SBJ**: "*librar**", "*TPL*", "*third-party*", or "*third party*;" *(iii)* **REC**: "*recommend**", "*suggest*", or "*assist*;" *(iv)* **TOOL**: "*tool**", "*approach*", "*system*", or "*methodology*".

[13] https://scopus.com

Table 3
Venues considered in the literature analysis.

	Acronym	Name
Conferences	ASE	The IEEE/ACM International Conference on Automated Software Engineering
	ESEC/FSE	The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
	ESEM	The ACM/IEEE International Symposium on Empirical Software Engineering and Measurement
	ICSE	The IEEE/ACM International Conference on Software Engineering
	ICSME	The International Conference on Software Maintenance and Evolution
	ICST	The IEEE International Conference on Software Testing, Verification and Validation
	ISSTA	The ACM SIGSOFT International Symposium on Software Testing and Analysis
	MSR	The IEEE International Working Conference on Mining Software Repositories
	SANER	The IEEE International Conference on Software Analysis, Evolution and Reengineering
Journals	EMSE	Springer Empirical Software Engineering Journal
	ESWA	Elsevier Expert Systems With Applications
	IST	Elsevier Information and Software Technology
	JSS	Elsevier Journal of Systems and Software
	TOSEM	ACM Transactions on Software Engineering and Methodology
	TSE	IEEE Transactions on Software Engineering

Listing 1: Excerpt of the Scopus query string.

```
TITLE-ABS ()(‘‘librar’’ OR ‘‘TPL’’ OR ‘‘third-party’’ OR ‘‘third party’’ OR ‘‘client’)
AND (‘‘migrat’’ OR ‘‘dependenc’’ OR ‘‘version’’ OR ‘‘evolve’’)
AND (‘‘recommend’’ OR ‘‘suggest’’ OR ‘‘assist’’)
AND (‘‘tool’’ OR ‘‘approach’’ OR ‘‘system’’ OR ‘‘methodology’ OR ‘‘framework’’)
AND (LIMIT-TO (PUBYEAR, 2022) OR [...]  OR  LIMIT-TO (PUBYEAR, 2017))
```

4.2.2. Code migration

Through a careful investigation, we found that although various code migration approaches do exist, e.g., MigrationMiner [34], SOAR [35] and [36], they are not eligible for the comparison with DeepMig. As shown in the motivating example in Fig. 1, DeepMig is different from existing approaches–especially the ones mentioned above–as it can provide API migrations at the definition level by cohesively considering the related TPLs. In fact, both SOAR [35,36] cannot perform migrations at the method level. Thus, no approach can be used for a direct comparison with DeepMig with respect to API migration.

We came across OpenRewrite,[14] an open source project for the automation of library migrations to enable large-scale distributed code refactoring for framework migrations, vulnerability patches, and API migrations in Java. It works by making changes to Abstract Syntax Trees (AST) representing source code and propagating the modified trees back to source code. It supports rewriting code by analyzing and transforming ASTs. OpenRewrite provides six predefined rewriting rules for TPLs, e.g., the migrations from `JUnit`[15] test assertions to `AssertJ`,[16] or from `log4j`[17] to `slf4j`[18] logging libraries. These migrations mainly contain changing statements and updating imports. The main drawback of rewriting code with OpenRewrite is that developers have to define specific rules to support migrations as shown in Fig. 9. In particular, Fig. 9(a) depicts the specification of migrating the dependencies, while Fig. 9(b) shows the rule to change `LogManager` to `LoggerFactory` API. Thus, this comparison aims to evaluate if DeepMig can automate API call migrations, overcoming the main difficulties caused by manually defined rules for cases that are not yet covered.

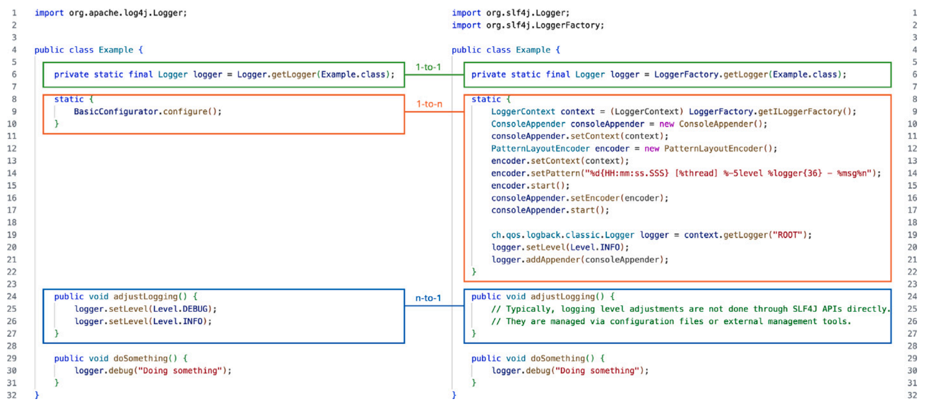
Large Language Models (LLMs) represent a cutting-edge approach in the automation of software development tasks [37], including the complex challenge of API function call migrations. LLMs such as ChatGPT are trained on diverse datasets encompassing a vast range of programming knowledge, which enables them to understand and generate code [38,39]. ChatGPT, in particular, can provide recommendations on migrating API function calls by leveraging its extensive training on code repositories, technical documentation, and forums [40]. ChatGPT operates by processing natural language and code inputs to generate human-like text responses. In the refactoring context, AlOmar et al. [41] studied the interactions between developers and ChatGPT concerning code refactoring. By analyzing 17,913 exchanges involving refactoring themes, the study seeks to understand how developers pinpoint areas for improvement and how ChatGPT meets their needs. This capability allows it to suggest code modifications, offer refactoring advice, and even write new code snippets that comply with the syntax and semantics of target APIs. For example, in migrating API function calls, ChatGPT can suggest changes similar to those between different logging frameworks — like converting `log4j` syntax to `slf4j` — based on its understanding of the libraries' documentation and usage patterns observed in its training data.

Unlike tools like OpenRewrite, which require predefined rules to perform migrations, LLMs (and ChatGPT) dynamically generate solutions based on the context provided during their interactions. However, the effectiveness of solutions provided by LLMs like ChatGPT can vary based on the specificity of the user queries and the model's exposure to similar problems during its training. Therefore, while LLMs offer a promising direction for automating code migration tasks, their success in specific instances depends on the users' detailed understanding and accurate expression of the problem context. When evaluating DeepMig against other code migration tools, we included ChatGPT (with its GPT-4 model) as a representative of language model-based approaches. We used its web chatbot interface with default settings and a temperature setting of 0.7 to balance creativity and fidelity in the model's responses. We also adhered to the system's default max tokens limit to ensure concise and contextually relevant responses. The model's task was to generate code migrations from prompts based on scenarios also used to evaluate DeepMig and OpenRewrite. This approach allowed us to assess how well a state-of-the-art language model supports code migration tasks compared to DeepMig, without requiring any specialized tuning or parameter adjustments. It provided insights into the out-of-the-box

[14] https://docs.openrewrite.org/
[15] http://www.junit.org
[16] https://joel-costigliola.github.io/assertj/
[17] https://logging.apache.org/log4j/2.x/
[18] https://www.slf4j.org/

Listing 2: An explanatory prompt for the illustrative migration task.

```
<s>
[INST] <<SYS>>You are a code assistant <</SYS>>
Given the following class implementation {{class_code}}
[/INST]
highlight the code changes needed to migrate from log4j to the slf4j library.
</s>
```



```
import org.apache.log4j.Logger;

public class Example {

    private static final Logger logger = Logger.getLogger(Example.class);

    static {
        BasicConfigurator.configure();
    }

    public void adjustLogging() {
        logger.setLevel(Level.DEBUG);
        logger.setLevel(Level.INFO);
    }

    public void doSomething() {
        logger.debug("Doing something");
    }
}
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Example {

    private static final Logger logger = LoggerFactory.getLogger(Example.class);

    static {
        LoggerContext context = (LoggerContext) LoggerFactory.getILoggerFactory();
        ConsoleAppender consoleAppender = new ConsoleAppender();
        consoleAppender.setContext(context);
        PatternLayoutEncoder encoder = new PatternLayoutEncoder();
        encoder.setContext(context);
        encoder.setPattern("%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n");
        encoder.start();
        consoleAppender.setEncoder(encoder);
        consoleAppender.start();

        ch.qos.logback.classic.Logger logger = context.getLogger("ROOT");
        logger.setLevel(Level.INFO);
        logger.addAppender(consoleAppender);
    }

    public void adjustLogging() {
        // Typically, logging level adjustments are not done through SLF4J APIs directly.
        // They are managed via configuration files or external management tools.
    }

    public void doSomething() {
        logger.debug("Doing something");
    }
}
```

Fig. 8. The Transformer-based recommendation engine.

utility of such models in software development contexts. In our evaluation, we fed ChatGPT with the zero-shot prompt strategy according to the query template reported in Listing 2. While we acknowledge that more advanced prompt strategies are in place [42,43], within the scope of this paper, we limit ourselves to the basic one.

To evaluate the GPT-4 migration capabilities, we adopted the following evaluation methodology. First, we used the DeepMig change detector to identify classes impacted by the migrations. Then, three co-authors with over 12 years of programming experience in Java asked ChatGPT to perform the migration using the prompt schema described in Listing 2. Each generated code migration was independently assessed by all the evaluators, and disagreements among all evaluators were collaboratively discussed until a consensus was reached. For each query, the involved authors annotated the suggested migration and manually computed precision and recall. In particular, we computed the number of true positives (TP), false positives (FP), and false negatives (FN). A TP occurs when there is an exact match at the signature level, i.e., both the recommended API and the ground-truth one are perfectly identical. Instead, a wrong code migration is considered an FP, while a code migration that differs from the ground truth is an FN.

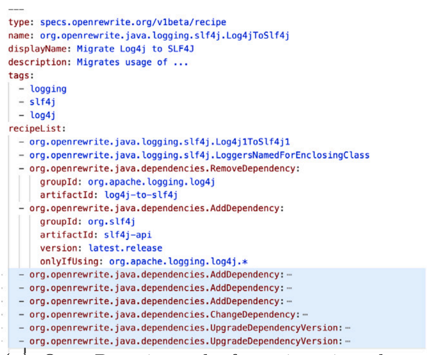
Upon evaluating API migration strategies, we encountered different types of function mappings. Fig. 8, shows simple one-to-one mappings occurring when replacing basic logging function calls, highlighted in green, such as switching from `log4j`'s `getLogger()` to `slf4j`'s `getLogger()`. More complex scenarios, such as one-to-many mappings highlighted in orange, occurred during configuration migration, where a single setup call in Log4j (e.g., BasicConfigurator.configure()) required multiple, more granular calls in `slf4j`/Logback. Additionally, many-to-one mappings, highlighted in blue, were seen when multiple direct log level adjustments in the old API were replaced by centralized configuration management in the new system. These mappings illustrate the varying complexity developers may face during API transitions.

It is important to point out that DeepMig processes a list of API function calls and generates the corresponding evolved versions, thus effectively handling the transformation from an old method definition to a new one (see Fig. 4). The quality and diversity of the training data significantly influence the tool's capability, allowing it to support any of the previously defined migration classes. Therefore, while DeepMig is designed to handle all types of migrations, including *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many*, its performance in terms of accuracy and computational efficiency is largely contingent on the diversity and richness of the dataset used to train the model.

4.3. Datasets

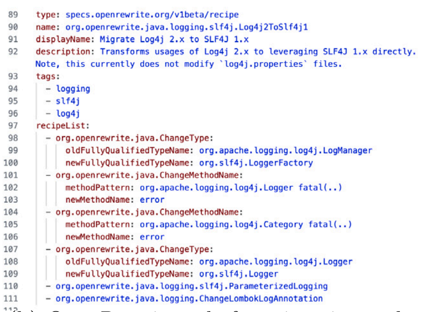
We collected three different datasets as explained below.

- **Library migration $\mathbf{D}_L$.** This is a dataset of 122,340 software projects with 35,543 TPLs has been curated through the process of mining the Maven Central Repository. To assess the performance of DeepMig, we collected only software projects that migrate, delete, add, or update at least one library between their versions. Then, for each collected software project, we only consider the consecutive version pairs that maximize the number of library migrations. $\mathbf{D}_L$ has been used to train DeepMig so that, given an input project, it can recommend for each depending TPL the alternative library to migrate to, if needed.
- **Code migration $\mathbf{D}_{C_R}$.** This dataset is used to compare DeepMig with OpenRewrite. As explained in Section 4.2, OpenRewrite supports the migration from `log4j` to `slf4j`, two popular logging libraries, among six predefined migrations. To provide input for the comparison, Lib Parser scanned the Maven Dependency Graph (MDG) consisting of 222,478 libraries, 2,407,335 total artifacts (including versions), and 9,715,699 relationships (including *next* and *depend* links) to identify artifact pairs that had migrated from `log4j` to `slf4j` at some point. Such a migration pair was considered owing solely to the fact that many clients replace `log4j` with `slf4j`. Lib Parser iterated through the MDG artifacts to identify libraries dependent on `log4j`. Upon identifying an artifact, the *next* relationships were inspected to detect any artifact pairs that removed the *dependent* link from `log4j` and added the *dependent* link to `slf4j`. Due to the considerable size and density of the MDG, Lib Parser operated continuously for 5 h to extract 50 artifact pairs. Upon thorough analysis of these collected pairs, all authors concurred that the collected artifact pairs exemplify representative migration cases from `log4j` to `slf4j` and agreed on the corpus size. As OpenRewrite works with source code, we manually linked the Maven artifacts to



```
---
type: specs.openrewrite.org/v1beta/recipe
name: org.openrewrite.java.logging.slf4j.Log4jToSlf4j
displayName: Migrate Log4j to SLF4J
description: Migrates usage of ...
tags:
  - logging
  - slf4j
  - log4j
recipeList:
  - org.openrewrite.java.logging.slf4j.Log4j1ToSlf4j1
  - org.openrewrite.java.logging.slf4j.LoggersNamedForEnclosingClass
  - org.openrewrite.java.dependencies.RemoveDependency:
      groupId: org.apache.logging.log4j
      artifactId: log4j-to-slf4j
  - org.openrewrite.java.dependencies.AddDependency:
      groupId: org.slf4j
      artifactId: slf4j-api
      version: latest.release
      onlyIfUsing: org.apache.logging.log4j.*
  - org.openrewrite.java.dependencies.AddDependency: …
  - org.openrewrite.java.dependencies.AddDependency: …
  - org.openrewrite.java.dependencies.AddDependency: …
  - org.openrewrite.java.dependencies.ChangeDependency: …
  - org.openrewrite.java.dependencies.UpgradeDependencyVersion: …
  - org.openrewrite.java.dependencies.UpgradeDependencyVersion: …
```

(a) OpenRewrite rule for migrating depedencies



```
type: specs.openrewrite.org/v1beta/recipe
name: org.openrewrite.java.logging.slf4j.Log4j2ToSlf4j1
displayName: Migrate Log4j 2.x to SLF4J 1.x
description: Transforms usages of Log4j 2.x to leveraging SLF4J 1.x directly.
Note, this currently does not modify `log4j.properties` files.
tags:
  - logging
  - slf4j
  - log4j
recipeList:
  - org.openrewrite.java.ChangeType:
      oldFullyQualifiedTypeName: org.apache.logging.log4j.LogManager
      newFullyQualifiedTypeName: org.slf4j.LoggerFactory
  - org.openrewrite.java.ChangeMethodName:
      methodPattern: org.apache.logging.log4j.Logger fatal(..)
      newMethodName: error
  - org.openrewrite.java.ChangeMethodName:
      methodPattern: org.apache.logging.log4j.Category fatal(..)
      newMethodName: error
  - org.openrewrite.java.ChangeType:
      oldFullyQualifiedTypeName: org.apache.logging.log4j.Logger
      newFullyQualifiedTypeName: org.slf4j.Logger
  - org.openrewrite.java.logging.slf4j.ParameterizedLogging
  - org.openrewrite.java.logging.ChangeLombokLogAnnotation
```

(b) OpenRewrite rule for migrating code

Fig. 9. Migration rules with OpenRewrite.

the corresponding version control system repositories. Since each specific code version needs to be linked onto the Maven artifact version, we had to filter out five projects as it was impossible to identify the version control storing the source code. Furthermore, five non-Maven GRADLE and ANT projects were removed. Among 40 migrations, OpenRewrite could not perform eleven of them: Three failed because of old Java versions (older than Java 1.8), while eight migrations were aborted because of missing dependencies or plugins. We obtained a set of clients for the experiments, as shown in Table 4. For each client shown in Table 4, the considered initial and target versions are reported in columns V_1 and V_2, respectively. The number of files that have been changed by OpenRewrite and DeepMig during migrations are reported in columns *#OpenRewrite files* and *#DeepMig files*, respectively. The number of files in the ground-truth, i.e., those that must be changed during the migrations are reported in the last column of the table.

- **Code migration $\mathbf{D}_C$ and $\mathbf{D}_{C_s}$.** The dataset $\mathbf{D}_C$ contains clients with migrations. Starting from the top-popular artifacts,[19] MDG was used to find as many as possible update pairs $\langle c_x, c_y \rangle$ where $AL_{\langle c_x,c_y \rangle} \neq \emptyset$ and $RL_{\langle c_x,c_y \rangle} \neq \emptyset$. This constraint allows us to mine update pairs that simulate possible migrations. Though we cannot assert that the update pairs are voluntary library migrations, they can include new and removed API calls. Eventually, LIB PARSER collected 3699 update pairs counting 16,850 migration pairs. To understand the impact of frequent definitions on DeepMig, $\mathbf{D}_{C_s}$ is extracted from $\mathbf{D}_C$ consisting of 3953 pairs where each definition occurs in at least 10 extracted clients.

4.4. Settings and metrics

Concerning the DeepMig hyperparameters, we considered the `vocabulary_size` (i.e., the total number of unique tokens that the model's embedding layer can handle), the `sequence_length` (i.e., the length of the input sequences that the model processes in each training step), and the `batch_size` (i.e., the number of training examples used in one forward/backward pass of the model). To ensure the robustness and validity of our Transformer model in practical applications, we set the hyperparameters as follows: a `vocabulary_size` of 1500, a `sequence_length` of 20, and a `batch_size` of 64. These settings were chosen based on their demonstrated effectiveness in balancing computational efficiency with model performance [8,44], which is crucial for real-world deployment.

Concerning the automated evaluation, we employed the ten-fold cross-validation methodology [45] so that we could test the approaches on the entire dataset and not on a (randomly chosen) test set.

Similarly to recent work [46], we consider *Success rate*, *Precision*, *Recall*, and the *Levenshtein edit distance* [47] to evaluate the results.

- For a testing project/definition t, the target set of TPLs, or the migrated definitions is called ground-truth data *GT(t)*.
- $REC(t)$ is the list of recommended TPLs/API calls for the testing project/definition t;
- $match(t) = GT(t) \bigcap REC(t)$ is the set of items in the recommended list that match with the ground-truth data.

We perform the evaluation of DeepMig and compare it with the baselines using the following metrics [46].

- **Success rate.** Given a set of T testing projects/definitions, this metric measures the rate at which a recommendation engine can return at least a match among the recommended TPLs/API calls for every testing instance $t \in T$. Success rate only evaluates how a model can provide a minimum number of matches, and thus it gives a rough reflection of the results, without saying how accurate the model is. This is why we require other metrics, as explained below.
- **Precision and Recall.** *Precision P* is the ratio of items matching with the ground-truth data to the total number of recommended items; i.e., N; *recall R* is the ratio of items that match with the ground-truth data to the total number of ground-truth items.
- **Levenshtein distance.** The Levenshtein edit distance [47] between two strings s_1 and s_2 counts the minimum number of insertions, deletions, and replacements (weighting 2) to obtain the second string from the first one [48]. This metric is used to measure the similarity between two sequences of artifacts. The distance between s_1 and s_2 corresponds to the number of substitutions needed to transform s_1 to s_2, defined as follows.[20]

$$L_{s_1,s_2}(i,j) = \begin{cases} max(i,j) & \text{if min(i,j)=0,} \\ min \begin{cases} L_{s_1,s_2}(i-1,j)+1 \\ L_{s_1,s_2}(i,j-1)+1 \\ L_{s_1,s_2}(i-1,j+1)+1 \end{cases} & \text{otherwise.} \end{cases} \quad (1)$$

In the evaluation, we measured the percentage of the testing instances for which the Levenshtein distance is equal to zero, i.e., when the recommended sequence and the ground-truth one perfectly match with each other, and we refer to this percentage as ρ_L.

5. Results

We analyze the experimental results obtained from the evaluation by answering the research questions in Section 4.1.

[19] https://mvnrepository.com/popular

[20] https://dzone.com/articles/the-levenshtein-algorithm-1

Table 4
Clients migrating from `log4j` to `slf4j` used for comparing DeepMig and OpenRewrite.

ID	Client	V_1	V_2	# OpenRewrite files	# DeepMig files	#GT files
C1	ai.api:libai	1.4.9	1.5.10	2	2	2
C2	au.csiro:snorocket-core	2.6.0	2.7.0	9	9	9
C3	be.objectify:objectify-led	1.0.1	1.2	4	3(a)	4
C4	ca.carleton.gcrc:nunaliit2-adhocQueries	2.0.1	2.0.2	2	2	2
C5	ca.carleton.gcrc:nunaliit2-auth-cookie	2.0.1	2.0.2	2	2	2
C6	ca.carleton.gcrc:nunaliit2-auth-http	2.0.1	2.0.2	4	4	4
C7	ca.carleton.gcrc:nunaliit2-contributions	2.0.1	2.0.2	5	5	5
C8	ca.carleton.gcrc:nunaliit2-dbSec	2.0.1	2.0.2	3(c)	4	4
C9	ca.carleton.gcrc:nunaliit2-dbWeb	2.0.1	2.0.2	1	1	1
C10	ca.carleton.gcrc:nunaliit2-jdbc	2.0.1	2.0.2	1	1	1
C11	ca.carleton.gcrc:nunaliit2-multimedia	2.0.1	2.0.2	13	13	13
C12	ca.carleton.gcrc:nunaliit2-onUpload	2.0.1	2.0.2	3	3	3
C13	ca.carleton.gcrc:nunaliit2-progress	2.0.1	2.0.2	2	2	2
C14	ca.carleton.gcrc:nunaliit2-search	2.0.1	2.0.2	2	2	2
C15	ca.carleton.gcrc:nunaliit2-upload	2.0.1	2.0.2	3	3	3
C16	ca.carleton.gcrc:nunaliit2-utils	2.0.1	2.0.2	1	1	1
C17	com.aerse:uploader	1.14	1.15	3	3	3
C18	com.llsfw:llsfw-core	2.2-RELEASE	2.3.1-RELEASE	11(c)	32	32
C19	com.ning:metrics.goodwill-access	0.1.3	0.1.4	3	1(d)	1
C20	com.omertron:fanarttvapi	1.4	1.5	7	3(a)	7
C21	com.omertron:themoviedbapi	3.3	3.4	42	39(a),(e)	42
C22	com.omertron:traileraddictapi	1.4	1.5	5	3(a),(e)	5
C23	com.pubnub:pubnub	3.4	3.5.4	0(b)	2	2
C24	net.anotheria:moskito-cdi	2.2.5	2.5.5	2	2	2
C25	org.apache.httpcomponents-client-rel	5.0-alpha2	5.0-beta1	10	4(a)	10
C26	org.apache.whirr:whirr-core	0.5.0-incubating	0.6.0-incubating	5	2(a)	5
C27	org.codemonkey.simplejavamail:simple-java-mail	2.1	2.2	1	1	1
C28	org.jscsi:target	2.2	2.4	17	16(a)	17
C29	uk.co.jemos.podam:podam	5.5.1.RELEASE	6.0.2.RELEASE	7	5(e)	7
			Total	**170**	**170**	**191**

(a) There are missing files as method signatures changed, e.g., the impacted method declarations have been renamed, moved, or removed; (b) OpenRewrite is not able to migrate Java files that do not belong to standard maven/gradle project structure; (c) It missed some Java files. By manually inspecting the code, we discovered that these files should be migrated; (d) OpenRewrite migrated unused imports. This means that the source code of these files does not invoke any *log4j* API call; (e) OpenRewrite also migrated test classes not included in the deployed jar, thus *Code Analyzer* cannot recognize these impacts.

Table 5
Number of papers for the related topics.

GOAL	SBJ	REC	TOOL
2,148	869	3,432	1,755
GOAL ∪ SBJ ∪ REC ∪ TOOL			
39			

5.1. **RQ**$_1$: How well does DeepMig perform in migrating libraries?

Searching for suitable baselines. The process described in Section 4.2.1 resulted in a corpus of 39 papers. After carefully reading the titles and abstracts of the resulting work, we selected 14 papers that appeared to propose approaches for dealing with the challenge of library migration. Finally, these papers are examined to identify potential baselines for comparison with DeepMig, and are reported as Table 5.

Møller and Torp [49] developed a model-based variant of type regression testing to find breaking changes by automatically generating tests from a reusable API model. This is not related to library migration as DeepMig does. SoftMon [50] is built on top of NLP techniques to compare the codebases of two separate applications to locate the exact set of functions that are disproportionately responsible for differences in performance. SoftMon was not conceived to migrate third-party libraries, and thus, it cannot be used as a baseline for the evaluation of DeepMig.

Chen et al. [51] presented an unsupervised deep learning-based approach to embed both API usage semantics and API description (name and document) semantics into vector space for mapping API between third-party libraries. Based on deep learning models trained using tens of millions of API call sequences, method names and comments of 2.8 million methods from 135,127 GitHub projects, the proposed tool obtains better performance than the baseline, i.e., a deep learning or traditional information retrieval (IR) methods for inferring likely analogical APIs.

Ochoa et al. [52] proposed an extended version of the japicmp tool, called Maracas, to detect breaking changes (BCs) between two versions of a TPL. To this end, the approach performs static analysis on the Java bytecode by considering a large corpus of Maven artifacts. Even though Maracas is similar to DeepMig, the tool is focused on identifying BCs rather than applying the actual migration.

In our prior work [8], we developed DeepLib to recommend TPL upgrading. The system analyzes migration history from several projects and predicts a set of future versions, leveraging long short-term memory recurrent networks. This is different from what is provided by DeepMig, whose recommendations contain both upgrading and migration.

Ponta et al. [53] introduced an approach to the identification, assessment, and mitigation of vulnerabilities in open-source software. Their approach takes a code-centric perspective and merges static and dynamic analyses to evaluate the accessibility of vulnerable segments within libraries employed by an application, whether directly or indirectly. By considering the usage context, the approach provides developers with valuable support in making informed decisions regarding non-vulnerable library versions. In the end, the approach by [53] is implemented as a tool named Vulas, that embodies the code-centric and usage-based methodology. Hence, whenever non-vulnerable library versions are accessible, an update to one of these versions stands as the recommended solution to rectify vulnerable application dependencies. In contrast to DeepMig, Vulas is different as it does not provide support for recommendations that encompass both the upgrading and migration aspects simultaneously.

LibHarmo [54] was proposed as an interactive, effort-aware technique to harmonize inconsistent library versions in Java Maven projects. LibHarmo's objective is to propose harmonized versions that require

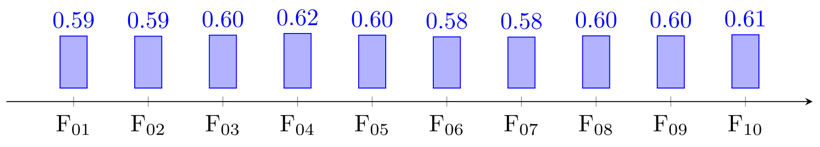


Fig. 10. RQ$_1$: Success rate on $\mathbf{D}_L$.

minimal harmonization efforts by considering factors like the reduction in the number of calls made to library APIs that experience deletions and modifications in the harmonized version. Similar to the other presented approaches, LibHarmo does not address the challenge of migrating from a library to a new one.

By reviewing the existing studies, we did not encounter any tools offering the exact type of recommendations provided by DeepMig. Thus, we paid attention to approaches that explicitly contain the word "*migration*" in their name. MigrationAdvisor [5] is the most recent tool, and it was compared with MigrationMiner [4,34,55] and [24]–two well-established baselines. Among others, we noticed that MigrationMiner can be an eligible tool for the evaluation of DeepMig. MigrationMiner is available in two versions, i.e., *(i)* a demo version accessible through a website; and *(ii)* a replication package. The former accepts an input query and retrieves directly the list of the most promising libraries. The latter is an offline version and provides a restful back-end. Since the training data used for the website is hidden, we cannot compare DeepMig with the online version. Next, we tried with the offline version which requires a local database. To conduct a fair comparison, using the same amount of input data is necessary. However, this is not possible as the format of the database is not disclosed. This is further confirmed when we carefully checked the threats to validity section and the GitHub documentation of MigrationMiner. Eventually, *we conclude that the comparison with MigrationMiner could not be done by any means.*

Summary. By thoroughly investigating the related work, we realize that there exist no tools that provide both library and code migration together as DeepMig does. While MigrationMiner is a potential baseline for library migration, the way it is released prevents us from performing a fair comparison with DeepMig.

Evaluation using a real-world dataset. To evaluate DeepMig, we ran it on $\mathbf{D}_L$, i.e., the curated dataset for library migration. For this comparison, we use success rate, precision, and recall as the evaluation metrics since they have been widely applied for this purpose [8].

The success rate scores obtained DeepMig for all the ten folds are shown in Fig. 10. Overall, the scores are always greater than 0.50, i.e., for F_{01}–F_{10}, success rates range from 0.58 to 0.62. This means that over half of the testing instances get at least a matched library. To further validate the performance, we consider the precision and recall scores obtained by DeepMig in Fig. 11. Through a manual analysis of a selection of recommended migrations, it appears that DeepMig suggests a larger number of concurrent libraries to be migrated compared to the libraries that a software project have been actually migrated. The motivation for this is mostly driven by the higher recall rates compared to precision.

Answer to RQ$_1$. DeepMig recommends suitable migrations for a complete set of libraries invoked by a project. More than a half of the projects under consideration get correct migration for at least a library.

5.2. *RQ$_2$: When can DeepMig generate API migration steps comparable to those recommended by OpenRewrite?*

To compare DeepMig and OpenRewrite, we ran them on $\mathbf{D}_{C_R}$ (a set of 29 migrating clients updating from `logj` to `slf4j`), and compared

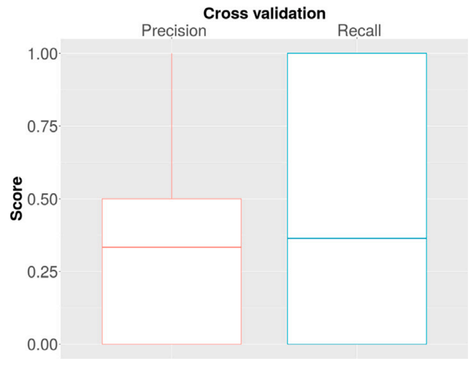


Fig. 11. RQ$_1$: Precision, recall on $\mathbf{D}_L$.

their results according to the metrics described in Section 4.4. Two co-authors of this paper — who have more than 12 years of programming experience — applied the OpenRewrite pre-defined rules meticulously. The code migration advised by OpenRewrite for all the clients was individually analyzed by each evaluator. In case there was inconsistency in the evaluation, we discussed together to reach a final consensus. A client is denoted as "*rewritten*" if its code has been successfully changed to the modification performed by OpenRewrite.

We counted the number of times OpenRewrite performs an accurate modification at the same code method where DeepMig recommends an update. The accuracy of a code change is determined by comparing each rewritten client with the authentic one downloaded from Maven. A true positive is recorded when there is an exact match at the signature level, i.e., both the recommended API and the ground-truth one are perfectly identical.

At the same time, we experimented DeepMig, i.e., for each trial, one client was used as a test set, while training data was extracted from the remaining 28 clients, measuring the performance of every impacted method. Table 6 reports the average precision and recall scores plus/minus their standard deviation (SD) for all the considered clients.[21] The table shows that OpenRewrite achieves an ideal performance, i.e., the majority of the scores are greater than 0.90, with many of them reaching 1.0. The performance of DeepMig is lower than that of OpenRewrite. Only for C26 DeepMig outperforms OpenRewrite, getting 0.87 as precision and recall.

We manually inspected the suggested migrations by DeepMig to understand which migration scenarios listed in Section 2.2 are supported. Given the high similarity of `log4j` and `slf4j`, there are a few distinct patterns that have been applied in migrating API function calls. A *one-to-one* scenario occurs when directly replacing a single API function, exemplified by substituting `log4j`'s `LogManager.getLogger()` with `slf4j`'s `LoggerFactory.getLogger()`. This transition represents a straightforward equivalence between the two libraries, providing a clear, direct mapping that maintains the same functionality with minimal complexity. Conversely, a *many-to-one* migration involves consolidating multiple operations into a single, more efficient function. For example, replacing the log4j call `log.debug(''..''.: '' + <any>)` with `slf4j`'s `log.debug(''...'', <any>)` is an example of the

[21] An SD equal to 0.00 means that there is only one impacted client.

Table 6
RQ_2: Precision and recall obtained with OpenRewrite (OR) and DeepMig (DM).

Client	C1		C2		C3		C4		C5	
Approach	DM	OR	DM	OR	DM	OR	DM	OR	DM	OR
Precision	0.35 ± 0.21	1.00 ± 0.00	0.22 ± 0.20	0.92 ± 0.08	0.50 ± 0.00	1.00 ± 0.00	0.75 ± 0.00	1.00 ± 0.00	0.75 ± 0.00	1.00 ± 0.00
Recall	0.64 ± 0.69	1.00 ± 0.69	0.32 ± 0.36	0.88 ± 0.12	0.88 ± 0.19	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
Client	C6		C7		C8		C9		C10	
Approach	DM	OR	DM	OR	DM	OR	DM	OR	DM	OR
Precision	0.75 ± 0.00	1.00 ± 0.00	0.37 ± 0.31	0.96 ± 0.08	0.44 ± 0.21	1.00 ± 0.00	0.75 ± 0.00	1.00 ± 0.00	0.57 ± 0.00	1.00 ± 0.00
Recall	1.00 ± 0.00	1.00 ± 0.00	0.43 ± 0.40	0.94 ± 0.12	0.48 ± 0.36	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.68 ± 0.16	1.00 ± 0.00
Client	C11		C12		C13		C14		C15	
Approach	DM	OR	DM	OR	DM	OR	DM	OR	DM	OR
Precision	0.67 ± 0.11	1.00 ± 0.00	0.23 ± 0.35	1.00 ± 0.00	0.37 ± 0.53	1.00 ± 0.00	0.62 ± 0.18	1.00 ± 0.00	0.67 ± 0.14	1.00 ± 0.00
Recall	0.98 ± 0.07	1.00 ± 0.00	0.27 ± 0.48	1.00 ± 0.00	0.50 ± 0.70	1.00 ± 0.00	0.80 ± 0.28	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
Client	C16		C17		C18		C19		C20	
Approach	DM	OR	DM	OR	DM	OR	DM	OR	DM	OR
Precision	0.75 ± 0.00	1.00 ± 0.00	0.44 ± 0.10	1.00 ± 0.00	0.36 ± 0.13	1.00 ± 0.00	0.50 ± 0.00	1.00 ± 0.00	0.58 ± 0.12	1.00 ± 0.00
Recall	1.00 ± 0.00	1.00 ± 0.00	0.83 ± 0.28	1.00 ± 0.00	0.42 ± 0.28	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.97 ± 0.04
Client	C21		C22		C23		C24		C25	
Approach	DM	OR	DM	OR	DM	OR	DM	OR	DM	OR
Precision	0.52 ± 0.09	0.99 ± 0.02	0.30 ± 0.27	1.00 ± 0.00	0.50 ± 0.00	0.87 ± 0.18	0.00 ± 0.00	1.00 ± 0.00	0.56 ± 0.21	1.00 ± 0.00
Recall	0.86 ± 0.16	0.99 ± 0.01	0.60 ± 0.54	0.98 ± 0.01	0.50 ± 0.00	0.83 ± 0.23	0.00 ± 0.00	1.00 ± 0.00	0.73 ± 0.31	1.00 ± 0.00
Client	C26		C27		C28		C29			
Approach	DM	OR	DM	OR	DM	OR	DM	OR		
Precision	**0.87 ± 0.18**	**0.32 ± 0.55**	0.50 ± 0.00	1.00 ± 0.00	0.47 ± 0.08	1.00 ± 0.00	0.40 ± 0.14	1.00 ± 0.00		
Recall	**0.87 ± 0.17**	**0.26 ± 0.46**	1.00 ± 0.00	1.00 ± 0.00	0.93 ± 0.17	1.00 ± 0.00	0.73 ± 0.36	1.00 ± 0.00		

many-to-one scenario. This migration not only replaces the logging function but also integrates string concatenation into the logging framework's variable substitution mechanism, reducing the need for explicit string operations and enhancing performance.

We explain the low accuracy of DeepMig by the data availability. Essentially, while OpenRewrite does not need any data for training as it relies only on fixed migration rules, DeepMig is highly dependent on existing updating API definitions from the training clients. Computing the Spearman's rank correlation coefficient index, we obtained ρ_P=0.1610 with p-value=0.018 and $\rho_R = 0.3149$ with $2.589e^{-6}$ for the correlation between the frequency of occurrences of the testing definitions and the obtained precision and recall scores, respectively. This indicates that there is a positive correlation between the amount of available training data and DeepMig's performance, i.e., the more data there is, the better accuracy DeepMig can gain. In this respect, we conclude that the recommendations provided by DeepMig will be improved if it is trained with more input data, covering several migration steps.

Answer to $\mathbf{RQ_2}$. In general, OpenRewrite works better compared to DeepMig, recommending more relevant migrations. Nevertheless, DeepMig can obtain a comparable recommendation when there is enough data available for learning.

5.3. $\mathbf{RQ_3}$: *What are the applicability and limitations of using ChatGPT for API migration?*

Using the same dataset in the previous research question, we investigate the capabilities of ChatGPT (with its GPT-4 model) to perform the same task covered by DeepMig and OpenRewrite. Similar to what was done before, we compute the metrics outlined in Section 4.4 using the evaluation process described in Section 4.2.2. Table 7 summarizes the obtained results. Overall, ChatGPT is able to perform the requested migration in most of the cases, i.e., the precision is equal to 1.00 for the majority of the examined clients. Nevertheless, the recall scores are low by various configurations, e.g., C8, C13, and C14 (to name a few), with a recall below 0.60. Moreover, we notice that ChatGPT fails in generating the migrated class for some client projects, i.e., C2, C25, C26, and C28. This is related to the number of tokens that ChatGPT can process for a single prompt, i.e., the GPT-4 model has a context window of 8 K tokens.[22] Remarkably, we observed that in one case, ChatGPT provides the wrong migration for the project `com.aerse:uploader:1.15`, i.e., it suggests Apache http instead of `slf4j`. Our intuition is that the peculiar context of the project leads to this erroneous generation. Thus, the active context plays an important role and needs to be carefully selected before querying the LLM. Notably, the recall score is negatively affected by a high number of FN. By carefully investigating those cases, we realize that ChatGPT migrates several slf4j using string formatting. For instance, the function `logger.error(‘‘Unable to parse image conversion threshold: ’’ + s);` is migrated with `logger.error(‘‘Unable to parse image conversion threshold: ’’, s);`. On the one hand, this is not a disruptive change since the semantics is not affected. On the other hand, developers' code that belongs to our ground-truth is not adopting such refactoring. Therefore, we mark this case as an FN in the scope of this evaluation.

Answer to $\mathbf{RQ_3}$. Our experiment shows that ChatGPT is capable of supporting the considered migration scenarios with high precision, albeit with low recall scores by various configurations. Moreover, it has some certain limitations in handling the application context due to the limited token window and decompiling issues.

5.4. $\mathbf{RQ_4}$: *How can DeepMig improve its performance?*

The results in RQ_1 indicate that while OpenRewrite outperforms DeepMig, our proposed approach can provide comparable recommendations to those of OpenRewrite for some clients. Nevertheless, the results achieved by DeepMig are considerably low in different testing instances. We conjecture that this happens due to the lack of training data. DeepMig requires API calls useful to the development scenario with code migration. Therefore, this research question analyzes under which conditions DeepMig can enhance its accuracy.

We evaluated DeepMig using $\mathbf{D}_C$ and $\mathbf{D}_{C_s}$. The former is a diverse dataset, as it contains migrations from several clients, while the latter is a subset of $\mathbf{D}_C$ *with more frequent definitions*. The use of these datasets

[22] https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4

Table 7
Precision and Recall scores obtained with the migrations performed by ChatGPT.

Client ID	# migrations	TP	FP	FN	Prec	Recall
C1	2	2	0	1	1.00	0.67
C2	0	ChatGPT is not able to provide any migrated code				
C3	8	6	0	2	1.00	0.75
C4	5	5	0	0	1.00	1.00
C5	4	4	0	0	1.00	1.00
C6	15	9	0	6	1.00	0.60
C7	10	8	0	2	1.00	0.80
C8	4	3	0	7	1.00	0.30
C9	1	1	0	0	1.00	1.00
C10	3	3	0	0	1.00	1.00
C11	20	12	0	8	1.00	0.60
C12	3	3	0	0	1.00	1.00
C13	4	2	0	2	1.00	0.50
C14	4	2	0	2	1.00	0.50
C15	7	4	0	3	1.00	0.57
C16	2	1	0	1	1.00	0.50
C17	7	4	2	1	0.67	0.80
C18	20	10	0	10	1.00	0.50
C19	3	3	0	0	1.00	1.00
C20	4	1	2	1	0.33	0.50
C21	27	27	0	0	1.00	1.00
C22	13	3	0	10	1.00	0.23
C.3	6	2	0	4	1.00	0.33
C24	2	2	0	0	1.00	1.00
C25	0	ChatGPT is not able to provide any migrated code				
C26	0	ChatGPT is not able to provide any migrated code				
C27	2	1	0	1	1	0.50
C28	0	ChatGPT is not able to provide any migrated code				
C29	8	6	0	2	1	0.75

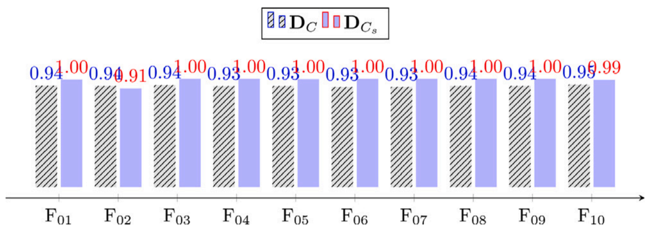


Fig. 12. RQ$_4$: Success rate for $\mathbf{D}_C$.

is to study the impact of frequently seen training data on the final performance. The success rate scores for all the ten folds, i.e., F_{01}–F_{10}, are shown in Fig. 12. DeepMig achieves very good performance across all the testing folds, i.e., the success rate scores for $\mathbf{D}_C$ are always larger than 0.93. This means that almost all the testing instances get at least a matched API. Such a performance is further improved with $\mathbf{D}_{C_s}$: Except for the last fold, DeepMig achieves a success rate of 1.0.

The precision and recall scores obtained from this experiment are shown using violin boxplots in Fig. 13. Such diagrams provide a more informative indication of the distribution's shape [56], highlighting the magnitude of the density. On $\mathbf{D}_C$, the scores in Fig. 13(a) range from 0.75 to 0.95, corresponding to a high prediction accuracy. By $\mathbf{D}_{C_s}$, the scores in Fig. 13(b) converge to the upper part of the diagram, and many of them stay close to the 1.0 level, implying an optimal performance.

Finally, we investigate whether the recommended set of API calls also matches the ground-truth data with respect to the order in which they appear. To this end, the percentage of recommendations getting 0 as Levenshtein distance (ρ_L) is computed and shown in Fig. 14. For $\mathbf{D}_C$, the scores range from 0.43 to 0.48, implying that nearly half of the testing definitions achieve a perfect match in the recommendations. ρ_L is greatly enhanced when we consider $\mathbf{D}_{C_s}$, i.e., apart from F_{02}, the scores are always greater than 0.84, reaching 0.90 as the maximum value. Altogether, we conclude that DeepMig considerably increases its performance even on a small dataset but with more frequent definitions ($\mathbf{D}_{C_s}$), compared to using a bigger dataset but with less frequent definitions ($\mathbf{D}_C$).

Answer to RQ$_4$. When it is fed with migrations executed by a large number of clients, DeepMig substantially improves its prediction, getting a perfect match for several testing instances.

6. Discussion

We discuss the main characteristics of DeepMig, including its limitations, and highlight the threats to the validity of the empirical evaluation we have performed to evaluate DeepMig.

6.1. The feasibility of DeepMig and its limitations

Based on the results of RQ$_2$, we noticed that the performance of DeepMig is inferior to that of OpenRewrite for several clients in our test set. This, however, does not harm the benefits of DeepMig. In fact, we do not try to claim that DeepMig can provide better migration plans compared to OpenRewrite. Compared to state-of-the-art approaches, DeepMig has the following benefits:

1. The value of DeepMig lies in its ability to *automate the migration*, i.e., without the need to develop any predefined rules. The latter is a drawback of OpenRewrite as it needs user-defined rewriting rules for each library migration. More importantly, as shown in the evaluation, when being fed with enough data, DeepMig is able to produce comparable recommendations with those of OpenRewrite. We faced difficulties in curating suitable data for the training, and this is the main reason why we did not get very good results with DeepMig on the considered datasets. We conjecture that if substantial effort is spent to collect data, our approach will be able to provide more accurate results.
2. Large Language Models (LLMs), including GPT-4, are resource-intensive, both computationally and financially. In order to train and test such models, one needs to rely on computing resources, or rely on third-party APIs, hence, in several scenarios, having to cope with privacy concerns. Instead, DeepMig is more lightweight, therefore both its training and inference require less resources than LLMs.

DeepMig is data-driven and its prediction capability relies heavily on *(i)* the availability; and *(ii)* the quality of the training data. This

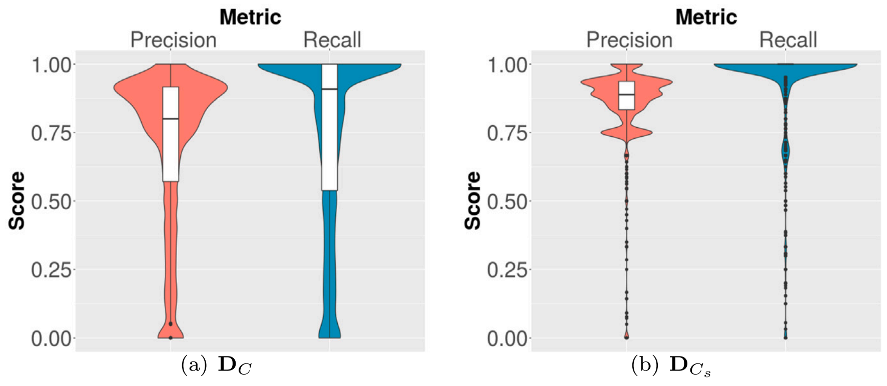


Fig. 13. RQ_4: Precision and Recall.

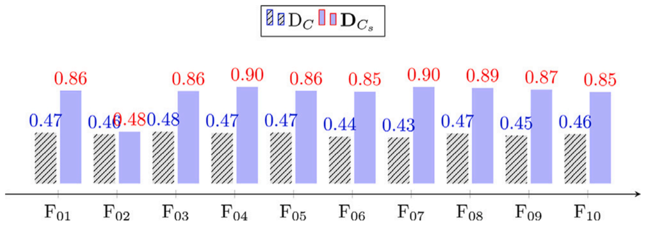


Fig. 14. RQ_4: Testing instances getting 0 as Levenshtein (ρ_L).

is confirmed by the outcomes of RQ_4, enforcing the view from RQ_2: DeepMig has the potential to boost its performance if it is fed with suitable data. Since DeepMig works on top of the Transformer, it properly captures the relationship among the API calls within a definition. Being transformer-based, DeepMig might further improve its performance if it is equipped with well-founded techniques developed for machine translation. Moreover, as reported in Section 4, we were able to collect the mapping only for method migrations, thus it can be confirmed that DeepMig is capable of handling specific categories of migrations, specifically those involving internal method invocations. The investigation did not include an examination of DeepMig's ability to execute various forms of migration, such as those that need modifications to the client project's architecture. Although DeepLib does not succeed on the TPL migration dataset, its potential should not be neglected. The tool works well with library upgrading [8], and we suppose that a fusion of DeepLib and DeepMig can be a practical solution to library migration and upgrading.

Concerning the experiment presented in RQ_3, we acknowledge that ChatGPT, as generic LLMs, can be used to cover the same tasks of DeepMig and OpenRewrite. However, we report the following limitations:

- **Fixed context.** As discussed in the experiment, ChatGPT and similar LLMs have a fixed input context window, which significantly restricts the amount of code or documentation they can analyze at one time. This limitation becomes particularly problematic in projects consisting of more than 10 files, each with hundreds of lines of code. The LLM cannot load and process huge contexts effectively, which can result in an incomplete understanding and subsequent analysis of the project, leading to potentially inaccurate assessments of impacted classes. We anticipate that this limitation can be handled by using fine-tuning as done in prior works [57–59]. We see this as a possible future work.
- **Decompiling issues.** LLMs, like ChatGPT, lack the capability to decompile code or directly inspect affected classes within a compiled binary. This inability restricts its use in scenarios where source code is not directly available or needs to be inferred from binaries, limiting its applicability in certain legacy systems or tightly compiled environments.

In this respect, we envision a combined usage of traditional tools such as DeepMig or OpenRewrite and cutting-edge generative AI models to overcome the identified issues.

6.2. Threats to validity

We identify the following threats that may affect our findings.

- Threats to *construct validity* concern the settings used to evaluate the systems, i.e., if they mimic a real application scenario. To this aim, we leverage data collected from real cases of migrations from Maven. Still, testing with data from other open sources that are popularly used by developers, e.g., GitHub, will help us further evaluate the considered systems. The selection of DeepLib as the baseline for TPL migration might be subject to external validity, i.e., the conclusions may or may not hold for other recommenders. We could not find any suitable baselines, as the most probable tools cannot be changed to work with other datasets. The comparison of DeepMig with OpenRewrite on the migration pairs from `logj` to `slf4j` may not be valid for other TPL pairs. We encountered difficulties in collecting data for other migration pairs supported by OpenRewrite as there are just a few clients performing them.
- Threats to *internal validity* are the confounding factors that might have an impact on the results. To compare with OpenRewrite, we directly executed the migrations through the available open-source project. This aims to remove any bias during the evaluation. Moreover, we also tried different sets of hyperparameters for training the Transformer, attempting to simulate real-world scenarios. In addition, we have performed a thorough analysis for hyperparameter calibration as explained in Section 4.4. However, we cannot exclude that there could be values for which DeepMig achieves better performances. Concerning RQ_2, the usage of ChatGPT may lead to incorrect migration due to the discussed limitations. To mitigate this, three co-authors manually checked the migration, computed the metrics, and reported any deviation from the expected output.
- Threats to *external validity* are the generalizability of our results with respect to additional datasets and programming languages. In this paper, we evaluated using datasets from Maven, and with source code written in Java. Our findings may not generalize to other types of artifacts out of the scope of our evaluation. Additional work is required to study the feasibility of DeepMig on other datasets and languages.

7. Related work

This section reviews the most related work by paying attention to library migration, code migration, and notable applications of transformer models in software engineering.

7.1. Library upgrading and migration

MigrationHelper [6] is a recent approach to the recommendation of library migrations. It formulates the task as a mining and ranking problem. Given a TPL, the system first mines target library candidates and then ranks them according to a combination of different designed metrics. SimilarAPI [60] employs an unsupervised RNN to recommend mapping between Java libraries by employing an API knowledge base extracted from GitHub repositories. Such data feeds the underpinning network to predict a ranked list of possible mappings given an initial TPL. [61] proposed EvoPlan, a system to provide upgrade plans for TPLs. EvoPlan exploits the experience of other projects that performed similar migrations to recommend the plan to consider when upgrading the current version to a more stable one. Similarly to EvoPlan, DeepLib [8] was conceived to recommend TPL upgrading. The system analyzes migration history from several projects and predicts a set of future versions, leveraging long short-term memory recurrent networks. SemDiff [62] is a tool that recommends adaptive changes at the API level. The system first analyzes the set of changes that have been performed in terms of methods by identifying the operations, e.g., deletion of methods. Afterward, a source repository was built to store the chain of method calls. SemDiff eventually exploits such data to adapt the underpinning project by considering the discovered API changes.

AURA [63] (AUtomatic change Rule Assistant) is an approach that combines text similarity with call dependency detection strategy to adapt a client project to a set of changes. Being built on top of a multi-iteration algorithm, the approach generates changing rules to apply the discovered changes.

LibSync [64] exploits graph-based data structures to identify and update APIs of two different library versions. Given the input project, the system identifies the map between the two APIs and retrieves the corresponding code snippets. Then, the graph-based representation is employed to capture the mutual relationships. Finally, LibSynch makes use of the learned adaptation patterns to recommend the locations and edit operations for adapting API usage.

Differently from DeepMig, these approaches do not recommend replacing a complete set of TPLs.

7.2. Automated approaches to support API migration

MigrationMiner [34] is an automated tool to detect code migrations between two Java TPLs. Given a project, it discovers any migration performed between the two TPLs and returns a collection of code changes, plus a set of related API documentation. Differently from DeepMig, MigrationMiner produces a list of code changes in a human-readable fashion. The code changes are not automatically performed: This task is left to the developer.

RAPIM [4] relies on semantic similarity to suggest method mappings during migration. The system employs text engineering techniques combined with SVM model to predict valid mappings given a pair of TPLs. M3 [65] supports the semantic-based migration of C libraries employing behavioral models. The CAnDL language is employed to discover patterns and apply API constraints to perform the actual migration. APIMigrator [16] exploits information in the mobile app's codebase to migrate API usage in Android apps. Given a target app and its description, the tool searches for the corresponding migration examples and transforms them into generic patches, which are eventually applied to perform the required migration using differential testing.

Patternika [66] is a practical solution to the automatic migration of APIs. The tool follows four main tasks: it mines AST-based patterns from library migration samples, automatically filters out irrelevant ones, manually creates and modifies migration patterns, and finally automatically applies patterns to patch the source code. SOAR [35] combines NLP models trained over API documentation with program synthesis to migrate and refactor APIs automatically. Patternika [66] is a practical solution to the automatic migration of APIs. The tool follows four main tasks: it mines AST-based patterns from library migrations samples, automatically filters out irrelevant ones, manually creates and modifies migration patterns, and finally automatically applies patterns to patch the source code. The migration capability was validated by testing different libraries, e.g., `slf4j` to `log4j`, or `java.utils` to `slf4j`, to cite a few. [36] employed two tools produced by Google, i.e., Error Prone and Refaster to refactor Java programs. To prove the approach's feasibility, the authors successfully migrated RxJava code to Reactor with a method coverage of 99%. DeepMig is different from these aforementioned approaches as it can provide API migrations at the definition level by taking into consideration the related TPLs in a cohesive manner. Both SOAR [35] and [36] cannot perform migration at the method level. A comparison with OpenRewrite demonstrated that DeepMig can generate comparable migration plans.

7.3. Application of transformer models in software engineering.

Recently, Transformer has gained traction from the SE community, being used in different tasks [67–69].

Mastropaolo et al. [57] investigate the usage of the T5 model to support four different SE tasks, i.e., automatic bug-fixing, injection of mutants, generation of assert statement, and code summarization. To enable such multi-task learning, the authors first preprocess four different datasets to fine-tune the original T5 model. Afterward, the overall performance of the enhanced model was been increased by varying the learning rate. The results show that the T5 is capable of outperforming existing baselines in each considered task.

Specifically designed for Python language, PYMT5 [70] is a text-to-text transfer transformer that predicts methods from natural language descriptions and summarizes code into docstrings. After a pre-training phase, the approach exploits the CodeT5 model to generate both method signatures and bodies.

IntelliCode Compose [71] exploits transformers to support multilingual code completion using a pre-trained multilayer generative model, namely GPT-C. First, the source code is encoded as a sequence of tokens by using a custom parser to create a vocabulary. Then, the model training is computed offline by exploiting a tailored data-parallel implementation to generate snippets of code. IntelliCode decodes the predicted sequences and generates snippets of code belonging to different languages, i.e., Javascript, Python, Typescript, and Javascript. Similarly, [72] present TravTrans, a GPT-2 transformer model that analyzes the syntactic structure to perform automated code completion. The system is capable of extracting path-based relations from the sequence of code tokens, thus predicting relevant code elements given an active context. A Transformer-Based Code Classifier (TBCC) was proposed to categorize code snippets by splitting the whole AST into multiple sub-trees [73]. The underpinning network makes use of attention mechanisms to *(i)* classify code snippets written in C; and *(ii)* detect software clones in Java snippets.

The most relevant work to DeepMig is CodeGen4Lib [74]. It has been proposed to support the generation of third-party libraries from the natural language description of the task. To this end, the approach generates two different types of library-related artifacts i.e., the import and the corresponding code snippets. To support the former type, the system exploits the BM25 information retrieval algorithm by relying on the initial query. The latter has been covered by using the CodeT5 transformer model. The automated evaluation confirms that CodeGen4Lib is suitable for generating high-quality code coherent with the generated libraries.

Overall, we can see that while transformer models have been applied in different domains, its applications in Software Engineering are still in their infancy. To the best of our knowledge, DeepMig is the first tool that employs a transformer model to recommend library and code migration.

8. Conclusion and future work

This paper presented a framework, named DeepMig, that supports the combined migration of Third-Party Libraries (TPLs) and API methods. Based on a literature review, we found that DeepMig is the first approach capable of recommending dual migration to software projects.

We have conducted an empirical evaluation aimed at assessing the recommendation performances achieved by DeepMig. The empirical study results show that:

1. DeepMig can recommend both TPL and API migration, providing developers with a practical tool to migrate entire projects.
2. On the evaluation dataset, over 50% of the studied projects get a correct migration for at least a library.
3. When being compared with a state-of-the-art tool OpenRewrite, DeepMig is outperformed by DeepMig for most of the clients because of the limited training available.
4. However, if the training set is enlarged, DeepMig is able to achieve nearly perfect predictions.

For future work, we plan to evaluate DeepMig with data coming from other ecosystems such as GitHub. Moreover, in the evaluation reported in this paper, we managed to collect the mapping only for method migrations. Therefore, we could only confirm that DeepMig can deal with this type of migration. We will investigate whether DeepMig can be applied to other types of migration. Moreover, we plan to investigate the applicability of DeepMig to different programming languages, given that they make available a tailored package manager. For instance, we can extend it to Python, for which we intend to support PyPI,[23] a premier Python library repository hosting over 500,000 third-party open-source packages and over 5,000,000 of library versions. We aim to crawl the most popular packages and analyze their interdependent relationships to assess the feasibility and effectiveness of our methodology in diverse ecosystems. Last but not least, we plan to improve the recommendation engine with other advanced machine translation techniques, e.g., investigate the combination of traditional techniques with LLMs, mitigating existing issues highlighted in both approaches.

CRediT authorship contribution statement

Juri Di Rocco: Writing – original draft, Validation, Software, Methodology, Data curation, Conceptualization. **Phuong T. Nguyen:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Claudio Di Sipio:** Writing – original draft, Validation, Software, Data curation. **Riccardo Rubei:** Writing – original draft, Validation, Software, Methodology, Data curation. **Davide Di Ruscio:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Funding acquisition, Conceptualization. **Massimiliano Di Penta:** Writing – review & editing, Writing – original draft, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The used data are available on Github.

[23] https://pypi.org/

Acknowledgments

This work has been partially supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY). The work has been also partially supported by the European Union–NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 PNRR *"FRINGE: context-aware FaiRness engineerING in complex software systEms"* grant n. P2022553SL. We acknowledge the Italian "PRIN 2022" project TRex-SE: *"Trustworthy Recommenders for Software Engineers"*, grant n. 2022LKJWHC. We thank the anonymous reviewers for their valuable comments and suggestions that helped us improve the paper.

References

[1] R.G. Kula, D.M. German, A. Ouni, T. Ishio, K. Inoue, Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration, Empir. Softw. Eng. 23 (1) (2018) 384–417, http://dx.doi.org/10.1007/s10664-017-9521-5.

[2] C. Vendome, M.L. Vásquez, G. Bavota, M. Di Penta, D.M. Germán, D. Poshyvanyk, When and why developers adopt and change software licenses, in: 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, 2015, pp. 31–40, http://dx.doi.org/10.1109/ICSM.2015.7332449.

[3] C. Vendome, D.M. Germán, M. Di Penta, G. Bavota, M.L. Vásquez, D. Poshyvanyk, To distribute or not to distribute?: why licensing bugs matter, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018, pp. 268–279, http://dx.doi.org/10.1145/3180155.3180221.

[4] H. Alrubaye, M.W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, J. Mcgoff, Learning to recommend third-party library migration opportunities at the API level, Appl. Soft Comput. 90 (2020) 106–140.

[5] H. He, Y. Xu, X. Cheng, G. Liang, M. Zhou, MigrationAdvisor: Recommending library migrations from large-scale open-source data, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings, ICSE-Companion, 2021, pp. 9–12, http://dx.doi.org/10.1109/ICSE-Companion52605.2021.00023.

[6] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, M. Zhou, A multi-metric ranking approach for library migration recommendations, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, 2021, pp. 72–83, http://dx.doi.org/10.1109/SANER50967.2021.00016.

[7] C. Teyton, J.-R. Falleri, X. Blanc, Automatic discovery of function mappings between similar libraries, in: 2013 20th Working Conference on Reverse Engineering, WCRE, 2013, pp. 192–201, http://dx.doi.org/10.1109/WCRE.2013.6671294.

[8] P.T. Nguyen, J. Di Rocco, R. Rubei, C. Di Sipio, D. Di Ruscio, DeepLib: Machine translation techniques to recommend upgrades for third-party libraries, Expert Syst. Appl. 202 (2022) 117267, http://dx.doi.org/10.1016/j.eswa.2022.117267, URL https://www.sciencedirect.com/science/article/pii/S0957417422006388.

[9] E. Derr, S. Bugiel, S. Fahl, Y. Acar, M. Backes, Keep me updated: An empirical study of third-party library updatability on android, in: B.M. Thuraisingham, D. Evans, T. Malkin, D. Xu (Eds.), ACM Conference on Computer and Communications Security, ACM, 2017, pp. 2187–2200, URL http://dblp.uni-trier.de/db/conf/ccs/ccs2017.html#DerrBFA017.

[10] Y. Duan, L. Gao, J. Hu, H. Yin, Automatic generation of non-intrusive updates for third-party libraries in android applications, in: 22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019, USENIX Association, 2019, pp. 277–292, URL https://www.usenix.org/conference/raid2019/presentation/duan.

[11] J. Huang, N. Borges, S. Bugiel, M. Backes, Up-to-crash: Evaluating third-party library updatability on android, in: 2019 IEEE European Symposium on Security and Privacy, EuroS P, 2019, pp. 15–30, http://dx.doi.org/10.1109/EuroSP.2019.00012.

[12] J. Visser, A. van Deursen, S. Raemaekers, Measuring software library stability through historical version analysis, in: Proceedings of the 2012 IEEE International Conference on Software Maintenance, ICSM '12, IEEE Computer Society, USA, 2012, pp. 378–387, http://dx.doi.org/10.1109/ICSM.2012.6405296.

[13] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, Y. Liu, An empirical study of usages, updates and risks of third-party libraries in java projects, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2020, pp. 35–45, http://dx.doi.org/10.1109/ICSME46990.2020.00014.

[14] A. Decan, T. Mens, E. Constantinou, On the impact of security vulnerabilities in the npm package dependency network, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 181–191, http://dx.doi.org/10.1145/3196398.3196401.

[15] N. Deshpande, M.W. Mkaouer, A. Ouni, N. Sharma, Search-based third-party library migration at the method-level, in: J.L. Jiménez Laredo, J.I. Hidalgo, K.O. Babaagba (Eds.), Applications of Evolutionary Computation, Springer International Publishing, Cham, 2022, pp. 173–190.

[16] M. Fazzini, Q. Xin, A. Orso, Apimigrator: An API-usage migration tool for android apps, in: Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems, MOBILESoft '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 77–80, http://dx.doi.org/10.1145/3387905.3388608.

[17] M. Lamothe, W. Shang, Exploring the use of automated API migrating techniques in practice: An experience report on android, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 503–514, http://dx.doi.org/10.1145/3196398.3196420.

[18] M. Lamothe, W. Shang, When APIs are intentionally bypassed: An exploratory study of api workarounds, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering, ICSE, 2020, pp. 912–924.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS '17, Curran Associates Inc., Red Hook, NY, USA, 2017, pp. 6000–6010.

[20] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, in: Y. Bengio, Y. LeCun (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015, URL http://arxiv.org/abs/1409.0473.

[21] J. Di Rocco, P.T. Nguyen, C. Di Sipio, R. Rubei, D. Di Ruscio, M. Di Penta, DeepMig: Supporting repository, 2023, https://github.com/MDEGroup/DeepMig/. (Online; Accessed 31 August 2023).

[22] D.L. Parnas, Information Distribution Aspects of Design Methodology, Tech. rep, Departement of Computer Science, Carnegie Mellon University, Pittsburgh, 1971.

[23] M.P. Robillard, What makes APIs hard to learn? Answers from developers, IEEE Softw. 26 (6) (2009) 27–34.

[24] C. Teyton, J.-R. Falleri, X. Blanc, Mining Library Migration Graphs, in: 2012 19th Working Conf. on Reverse Engineering, 2012, pp. 289–298, http://dx.doi.org/10.1109/WCRE.2012.38.

[25] H. Alrubaye, M.W. Mkaouer, A. Ouni, On the use of information retrieval to automate the detection of third-party java library migration at the method level, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension, ICPC, 2019, pp. 347–357, http://dx.doi.org/10.1109/ICPC.2019.00053.

[26] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, O. Barais, The maven dependency graph: a temporal graph-based representation of maven central, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories, MSR, IEEE, 2019, pp. 344–348.

[27] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M.J. Steindorfer, J.J. Vinju, M3: A general model for code analytics in rascal, in: 1st International Workshop on Software Analytics, IEEE, Piscataway, 2015, pp. 25–28, http://dx.doi.org/10.1109/SWAN.2015.7070485.

[28] J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: ASE, ACM, 2014, pp. 313–324.

[29] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.

[30] B.A. Kitchenham, P. Brereton, Z. Li, D. Budgen, A.J. Burn, Repeatability of systematic literature reviews, in: 15th International Conference on Evaluation & Assessment in Software Engineering, EASE 2011, Durham, UK, 11-12 April 2011, Proceedings, 2011, pp. 46–55, http://dx.doi.org/10.1049/ic.2011.0006.

[31] S.G. MacDonell, M.J. Shepperd, B.A. Kitchenham, E. Mendes, How reliable are systematic reviews in empirical software engineering? IEEE Trans. Softw. Eng. 36 (5) (2010) 676–687, http://dx.doi.org/10.1109/TSE.2010.28.

[32] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: M.J. Shepperd, T. Hall, I. Myrtveit (Eds.), 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014, ACM, 2014, pp. 38:1–38:10, http://dx.doi.org/10.1145/2601248.2601268.

[33] H. Zhang, M.A. Babar, P. Tell, Identifying relevant studies in software engineering, Inf. Softw. Technol. 53 (6) (2011) 625–637.

[34] H. Alrubaye, M.W. Mkaouer, A. Ouni, MigrationMiner: An automated detection tool of third-party java library migration at the method level, in: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2019, pp. 414–417, http://dx.doi.org/10.1109/ICSME.2019.00072.

[35] A. Ni, D. Ramos, A.Z.H. Yang, I. Lynce, V. Manquinho, R. Martins, C. Le Goues, SOAR: A synthesis approach for data science API refactoring, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, 2021, pp. 112–124, http://dx.doi.org/10.1109/ICSE43902.2021.00023.

[36] R. Ossendrijver, S. Schroevers, C. Grelck, Towards automated library migrations with error prone and refaster, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, SAC '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1598–1606, http://dx.doi.org/10.1145/3477314.3507153.

[37] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, J.M. Zhang, Large language models for software engineering: Survey and open problems, in: 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 31–53, http://dx.doi.org/10.1109/ICSE-FoSE59343.2023.00008, URL https://doi.ieeecomputersociety.org/10.1109/ICSE-FoSE59343.2023.00008.

[38] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, B. Myers, Using an LLM to help with code understanding, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024, http://dx.doi.org/10.1145/3597503.3639187.

[39] P. Vaithilingam, T. Zhang, E.L. Glassman, Expectation vs experience: Evaluating the usability of code generation tools powered by large language models, in: Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, in: CHI EA '22, Association for Computing Machinery, New York, NY, USA, 2022, http://dx.doi.org/10.1145/3491101.3519665.

[40] J. Liu, C.S. Xia, Y. Wang, L. Zhang, Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, Adv. Neural Inf. Process. Syst. 36 (2024).

[41] E.A. AlOmar, A. Venkatakrishnan, M.W. Mkaouer, C.D. Newman, A. Ouni, How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations, in: 2024 IEEE/ACM 21st International Conference on Mining Software Repositories, MSR, 2024, pp. 202–206.

[42] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E.H. Chi, Q.V. Le, D. Zhou, Chain-of-thought prompting elicits reasoning in large language models, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, la, USA, November 28 - December 9, 2022, 2022, URL http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

[43] X. Li, S. Yuan, X. Gu, Y. Chen, B. Shen, Few-shot code translation via task-adapted prompt learning, J. Syst. Softw. 212 (2024) 112002, http://dx.doi.org/10.1016/j.jss.2024.112002, URL https://www.sciencedirect.com/science/article/pii/S0164121224000451.

[44] S. Casola, I. Lauriola, A. Lavelli, Pre-trained transformers: an empirical comparison, Mach. Learn. Appl. 9 (2022) 100334, http://dx.doi.org/10.1016/j.mlwa.2022.100334, URL https://www.sciencedirect.com/science/article/pii/S2666827022000445.

[45] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: 14th International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, 1995, pp. 1137–1143.

[46] P.T. Nguyen, J.D. Rocco, C.D. Sipio, D.D. Ruscio, M.D. Penta, Recommending API function calls and code snippets to support software development, IEEE Trans. Softw. Eng. (2021) 1, http://dx.doi.org/10.1109/TSE.2021.3059907.

[47] V. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, Soviet Phys. Doklady 10 (1966) 707.

[48] N. Babar, Levenshtein algorithm, https://dzone.com/articles/the-levenshtein-algorithm-1. (Online; Accessed 21 August 2023).

[49] A. Møller, M.T. Torp, Model-based testing of breaking changes in node.js libraries, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 409–419, http://dx.doi.org/10.1145/3338906.3338940.

[50] S.S. Singh, S.R. Sarangi, SoftMon: A tool to compare similar open-source software from a performance perspective, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 397–408, http://dx.doi.org/10.1145/3379597.3387444.

[51] C. Chen, Z. Xing, Y. Liu, K.O.L. Xiong, Mining likely analogical APIs across third-party libraries via large-scale unsupervised API semantics embedding, IEEE Trans. Softw. Eng. 47 (3) (2021) 432–447, http://dx.doi.org/10.1109/TSE.2019.2896123.

[52] L. Ochoa, T. Degueule, J.-R. Falleri, J. Vinju, Breaking bad? Semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study, Empirical Softw. Engg. 27 (3) (2022) http://dx.doi.org/10.1007/s10664-021-10052-y.

[53] S.E. Ponta, H. Plate, A. Sabetta, Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, IEEE Computer Society, 2018, pp. 449–460, http://dx.doi.org/10.1109/ICSME.2018.00054.

[54] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, X. Peng, Interactive, effort-aware library version harmonization, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 518–529, http://dx.doi.org/10.1145/3368089.3409689.

[55] H. Alrubaye, M.W. Mkaouer, A. Ouni, On the use of information retrieval to automate the detection of third-party java library migration at the method level, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension, ICPC, 2019, pp. 347–357, http://dx.doi.org/10.1109/ICPC.2019.00053.

[56] J.L. Hintze, R.D. Nelson, Violin plots: A box plot-density trace synergism, Amer. Statist. 52 (2) (1998) 181–184, http://dx.doi.org/10.1080/00031305.1998.10480559, arXiv:https://amstat.tandfonline.com/doi/pdf/10.1080/00031305.1998.10480559, URL https://amstat.tandfonline.com/doi/abs/10.1080/00031305.1998.10480559.

[57] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader Palacio, D. Poshyvanyk, R. Oliveto, G. Bavota, Studying the usage of text-to-text transfer transformer to support code-related tasks, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, 2021, pp. 336–347, http://dx.doi.org/10.1109/ICSE43902.2021.00041.

[58] F. Zhang, B. Chen, Y. Zhao, X. Peng, Slice-based code change representation learning, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, 2023, pp. 319–330, http://dx.doi.org/10.1109/SANER56733.2023.00038.

[59] A. Zlotchevski, D. Drain, A. Svyatkovskiy, C.B. Clement, N. Sundaresan, M. Tufano, Exploring and evaluating personalized models for code generation, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in: ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1500–1508, http://dx.doi.org/10.1145/3540250.3558959, URL https://doi-org.univaq.idm.oclc.org/10.1145/3540250.3558959.

[60] C. Chen, SimilarAPI: Mining Analogical APIs for Library Migration, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2020, pp. 37–40, ISSN: 2574-1926.

[61] R. Rubei, D. Di Ruscio, C. Di Sipio, J. Rocco, P. Nguyen, Providing upgrade plans for third-party libraries: A recommender system using migration graphs, Appl. Intell. 52 (2022) http://dx.doi.org/10.1007/s10489-021-02911-4.

[62] B. Dagenais, M.P. Robillard, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 19:1–19:35.

[63] W. Wu, Y. Guéhéneuc, G. Antoniol, M. Kim, AURA: a hybrid approach to identify framework evolution, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, ACM, 2010, pp. 325–334.

[64] H.A. Nguyen, T.T. Nguyen, G.W. Jr., A.T. Nguyen, M. Kim, T.N. Nguyen, A graph-based approach to API usage adaptation, in: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA.

[65] B. Collie, P. Ginsbach, J. Woodruff, A. Rajan, M.F. O'Boyle, M3: Semantic API migrations, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2020, pp. 90–102.

[66] E. Blech, A. Grishchenko, I. Kniazkov, G. Liang, O. Serebrennikov, A. Tatarnikov, P. Volkhontseva, K. Yakimets, Patternika: A pattern-mining-based tool for automatic library migration, in: IEEE International Symposium on Software Reliability Engineering, ISSRE 2021 - Workshops, Wuhan, China, October 25-28, 2021, IEEE, 2021, pp. 333–338, http://dx.doi.org/10.1109/ISSREW53611.2021.00098.

[67] C. Watson, N. Cooper, D.N. Palacio, K. Moran, D. Poshyvanyk, A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research, ACM Trans. Softw. Eng. Methodol. 31 (2) (2022) 32:1–32:58, http://dx.doi.org/10.1145/3485275, URL https://dl.acm.org/doi/10.1145/3485275.

[68] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large Language Models for Software Engineering: A Systematic Literature Review, 2023, http://dx.doi.org/10.48550/arXiv.2308.10620, URL arXiv:2308.10620[cs].

[69] R. Tufano, L. Pascarella, G. Bavota, Automating Code-Related Tasks Through Transformers: The Impact of Pre-training, 2023, http://dx.doi.org/10.48550/arXiv.2302.04048, URL arXiv:2302.04048[cs].

[70] C.B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, N. Sundaresan, PyMT5: multi-mode translation of natural language and Python code with transformers, 2020, http://dx.doi.org/10.48550/arXiv.2010.03150, URL arXiv:2010.03150[cs].

[71] A. Svyatkovskiy, S.K. Deng, S. Fu, N. Sundaresan, IntelliCode compose: Code generation using transformer, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1433–1443.

[72] S. Kim, J. Zhao, Y. Tian, S. Chandra, Code prediction by feeding trees to transformers, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, 2021, pp. 150–162, http://dx.doi.org/10.1109/ICSE43902.2021.00026.

[73] W. Hua, G. Liu, Transformer-based networks over tree structures for code classification, Appl. Intell. 52 (8) (2022) 8895–8909, http://dx.doi.org/10.1007/s10489-021-02894-2, URL https://link.springer.com/10.1007/s10489-021-02894-2.

[74] M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, X. Peng, CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation, 2023.