MELT: Mining Effective Lightweight Transformations from Pull Requests

Daniel Ramos
School of Computer Science, INESC-ID
Carnegie Mellon University, USA
drramos@scs.cmu.edu

Hailie Mitchell
Computer Science Department
Dickinson College, USA
mitchelh@dickinson.edu

Inês Lynce
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa, Portugal
ines.lynce@tecnico.ulisboa.pt

Vasco Manquinho
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa, Portugal
vasco.manquinho@inesc-id.pt

Ruben Martins
School of Computer Science
Carnegie Mellon University, USA
rubenm@cs.cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University, USA
clegoues@cs.cmu.edu

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00117

***Abstract*—Software developers often struggle to update APIs, leading to manual, time-consuming, and error-prone processes. We introduce MELT, a new approach that generates lightweight API migration rules directly from pull requests in popular library repositories. Our key insight is that pull requests merged into open-source libraries are a rich source of information sufficient to mine API migration rules. By leveraging code examples mined from the library source and automatically generated code examples based on the pull requests, we infer transformation rules in `Comby`, a language for structural code search and replace. Since inferred rules from single code examples may be too specific, we propose a generalization procedure to make the rules more applicable to client projects. MELT rules are syntax-driven, interpretable, and easily adaptable. Moreover, unlike previous work, our approach enables rule inference to seamlessly integrate into the library workflow, removing the need to wait for client code migrations. We evaluated MELT on pull requests from four popular libraries, successfully mining 461 migration rules from code examples in pull requests and 114 rules from auto-generated code examples. Our generalization procedure increases the number of matches for mined rules by 9×. We applied these rules to client projects and ran their tests, which led to an overall decrease in the number of warnings and fixing some test cases demonstrating MELT's effectiveness in real-world scenarios.**

I. Introduction

Developers often make use of third-party libraries [1], which provide modular functionality to clients through an Application Programming Interface (API). The API is a contract between the library and its clients, separating the concrete implementation of library features from its specification.

Ideally, APIs should remain stable. However, they change frequently, and API contracts are often broken [2], [3], [4], for reasons ranging from bug fixes, to changes in library requirements [5]. APIs may become deprecated or obsolete [6], requiring clients to adapt their code to reflect the newest library version. These kinds of non-functional code changes are known as *software refactoring* [7], a primarily manual [8] and error-prone [9] task. To migrate to a new library version, clients must examine the library changes such as by inspecting documentation or source code. This task's complexity often deters library clients from updating altogether, despite the security risks posed by outdated dependencies [10] [11].

The widespread prevalence of deprecations and breaking changes in the software ecosystem motivates research efforts in automating migration [12], [13], [14], [15], [16], [17]. Tools for API migration typically either mine commits from library client projects that have undergone migrations [15], [16], [12], or are supplemented by information from new client projects in the most up-to-date APIs [14]. The effectiveness of these tools is hindered by their reliance on mining data from client projects that have either already migrated across versions, or are already using up-to-date APIs. Unfortunately, this data is scarce: a recent study found that 81.5% [18] of projects keep outdated dependencies. Additionally, the mining process can only occur after clients begin transitioning between versions, precluding use shortly after a new version of the library is released [19], [20].

To overcome these limitations, we propose a new approach called MELT. Unlike previous methods, MELT does not require external data from clients. Instead, it leverages the fact that the development process of open-source libraries provides a wealth of high-quality information that is sufficient to generate transition examples and mine transformation rules. At a high level, *our idea is to use pull requests (PRs) submitted to a library's repository to learn code transformation rules for updating client code*. This allows the integration of transformation rule mining into the development process.

Pull requests have become the *de facto* standard for open-source software development on collaborative platforms like GitHub [21], [22]. Pull requests typically include a title, a natural language description of the proposed changes and how they relate to project milestones or issues, and a set of commits (i.e., code file changes). These are reviewed by a core group of maintainers who determine to accept, request revisions, or reject the changes.

We use information from pull requests merged into open-source libraries to mine transformation rules that adapt client code in light of breaking changes or deprecations. First, MELT

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00117

uses natural language descriptions from PRs to identify API changes by searching for keywords such as “deprecated”, “breaking change”, and “API change”. If the PR corresponds to such an API change, MELT first *uses the commits in the PRs that contain changes to the library code to generate transformation rules*. The internal updates to the library source code and test cases serve as the ground truth for mining transformation rules.

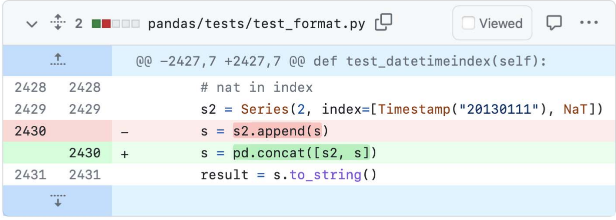
However, the code-level changes alone do not always provide sufficient information to mine thorough transformation rules for a given breaking change. MELT therefore additionally *leverages the natural language text in pull requests to generate additional code examples for mining.* Specifically, we prompt a state-of-the-art large language model (LLM) pre-trained on open-source code to generate concise code examples that clearly illustrate how to transition from the old API to the new one. Using its prior knowledge of the library (obtained from pretraining) and the additional information in the prompt, the model can often infer how to transition from old APIs and provide useful concrete examples.

Using the code examples mined from the library source and the code examples automatically generated from the natural language descriptions, we infer transformation rules in Comby [23], [24], a tool and a language for structural code search and replace. We choose to represent our transformation rules in Comby because: (1) it allows us to express find-and-replace rules in a concise and interpretable format; (2) it is a stable widely adopted tool for syntax-driven code transformations.

There are multiple key advantages to our approach. Firstly, MELT does not require client projects to infer transformation rules. This contrasts with previous work [25], [26], [27], [28], which require large datasets of training data containing multiple migration examples to mine rules. Indeed, MELT can mine migration rules even for changes that have not been in a library release yet (i.e., they are due to future milestones). Secondly, MELT uses Comby to express migration rules, which results in easy-to-interpret, adaptable, and maintainable transformations.

In summary, our main contributions are as follows:

- We introduce a novel approach to extract rules that address deprecations and breaking changes in open-source software libraries that does not rely on client data.
- An LLM-driven approach for generating code examples and test cases for transformation rule mining.
- A generalization procedure for transformation rules enhances their applicability in client projects.
- To facilitate integration into existing workflows, we prototype a continuous integration (CI) solution using GitHub Actions for library maintainers, so they can integrate MELT in their workflows.[1]
- We evaluate MELT on four open-source libraries to infer a total of 461 migration rules from code examples and 114 from auto generated code examples. We also evaluate MELT end-to-end by migrating client code.



```
@@ -2427,7 +2427,7 @@ def test_datetimeindex(self):
            # nat in index
            s2 = Series(2, index=[Timestamp("20130111"), NaT])
-           s = s2.append(s)
+           s = pd.concat([s2, s])
            result = s.to_string()
```

Fig. 1: Code change in pull request #44539 [29] from the pandas-dev/pandas repository.

TABLE I: *Top:* Comby rules extracted from pandas pull request #44539, deprecating DataFrame.append and Series.append. *Bottom:* Rules extracted from sci-py pull request #14419, including original specific (“Spec”) and generalized (“Gen”) versions. Template variable constraints are omitted for brevity.

	Match Template	Rewrite Template

	`:[[s2]].append(:[[s1]])`
	`where :[[s1]].type == Series`
	`:[[s2]].type == Series`
		`pd.concat([:[[s2]], :[[s1]]])`

	`:[[df]].append(:[[s]])`
	`where :[[df]].type == DataFrame`
	`:[[s]].type == Series`
		`pd.concat([:[[df]], DataFrame(:[[s]]).T. infer_objects()])`

Type	Match Template	Rewrite Template
Spec	`:[[s]].spline. cspline2d(:[[x]],:[y])`	`:[[s]].cspline2d( :[[x]], :[y])`
Gen	`:[[s]].spline cspline2d(:[args])`	`:[[s]].cspline2d( :[args])`

II. MOTIVATION AND OVERVIEW

Figure 2 provides a high-level overview of MELT and its main components. We delve into the specifics of each component in Sections III and IV.

Pull requests are the input of MELT, as they are the key source that informs our approach. Pull requests generally contain all the code changes related to a given new feature. For example, Figure 1 shows an example code change from a pull request [29] submitted to pandas [30] that deprecates two popular APIs: DataFrame.append and Series.append.[2] MELT identifies code changes, such as the one shown in Figure 1, within the pull request using its *Code Change Analyzer* (Section III-A) and inputs them into the *Rule Inference* algorithm (Section IV-A) to generate rules. The top portion of Table I shows two of the rules MELT infers from the code changes for this specific pull request.

The rules in Table I are expressed in Comby’s domain specific language [24]. The match template (left column) is the code structure for which Comby searches. The rewrite template (right column) shows how to transform the matched code based on the variables in the match template [23]. Comby uses template variables, i.e., placeholders that can be matched with certain

[1] https://github.com/squaresLab/melt_action

[2] Both APIs were later removed from pandas in version 2.0.0.

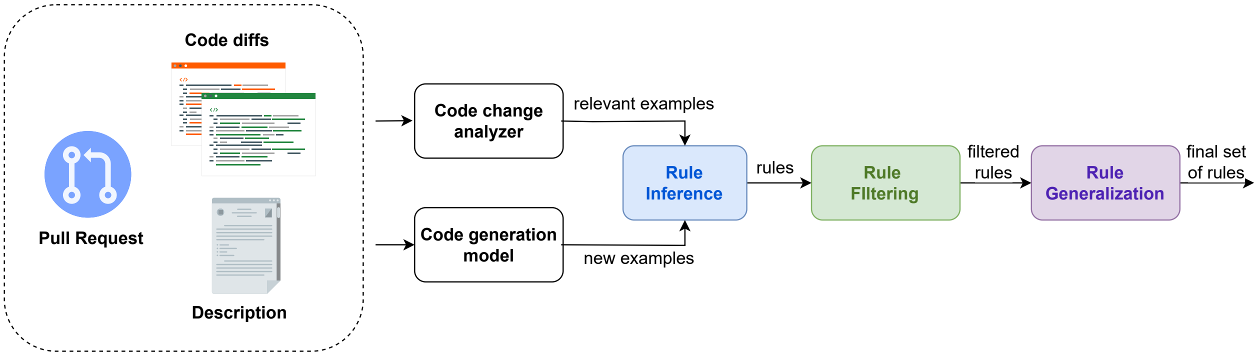
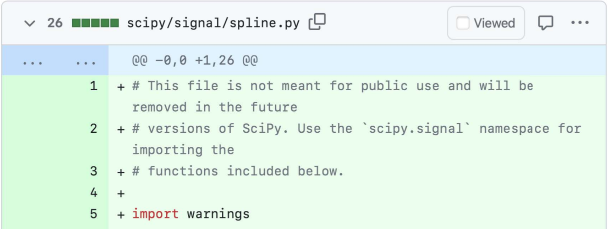


Fig. 2: MELT overview. MELT takes as input a pull request (PR) and outputs a set of rules. The PR is processed in two ways: (1) the Code change analyzer identifies relevant code changes; (2) the Code generation model generates additional code examples. Rules are inferred from the code changes and examples using the rule inference algorithm, then filtered and generalized.



```
@@ -0,0 +1,26 @@
+ # This file is not meant for public use and will be
  removed in the future
+ # versions of SciPy. Use the `scipy.signal` namespace for
  importing the
+ # functions included below.
+
+ import warnings
```

Fig. 3: Pull Request #14419 [33] from scipy/scipy. This pull request was part of SciPy 1.8, released in February 2022.

```
def old_usage1(image):
  return signal.spline.cspline2d(image, 8.0)

def new_usage1(image):
  return signal.cspline2d(image, 8.0)

class TestEquiv(unittest.TestCase):
  def test_assert1(self):
    np.random.seed(181819142)
    image = np.random.rand(71, 73)
    assert np.allclose(
            old_usage1(image),
            new_usage1(image))
```

Fig. 4: Code generated by GPT-4 showing how to transition from the deprecated namespace for cspline2d and a test case.

language constructs. For example, a template variable to match alphanumeric characters is represented by :[[x]], where x is the name of the template variable. The template variables in the match template can be constrained in multiple ways using a where clause. In particular, to prevent spurious matches, template variables can be constrained to be a certain type (like :[[s2]].type == DataFrame). Although type information is not strictly required, it is especially useful when working with common API names such as append and concat, since both are part of Python's builtins list.

Code diffs in pull requests provide valuable information, however, they do not always contain the necessary code examples for rule inference. Fortunately, pull requests offer alternative sources of information that can be used to extract further details about the changing APIs. Figure 3 shows an informative comment left by a developer in a code file when deprecating namespace scipy's [31] namespace scipy.signal.spline in favor of scipy.signal. To leverage all available information in the pull request, MELT uses a *Code Generation Model* to generate additional code examples and test cases for this change (Section III-B). Figure 4 shows a simplified version of code GPT-4 [32] (a state-of-the-art model) generates from the pull request in Figure 3. The generated examples enable us to both infer and test the rules.

Since the test case executes successfully, MELT uses the code example to generate a rule by abstracting concrete identifiers and literals. For this case, MELT generates the rule in the third row of Table I. This rule accurately reflects the deprecation made in the pull request (i.e., replaces the deprecated namespace with the new one). Nevertheless, a closer inspection reveals that the rule is too specific: it will only match usages where: (1) the first argument of cspline2d is an identifier (:[[s]] only matches with identifiers), and (2) the function is called with two or more arguments. The cspline2d function can accept multiple combinations of arguments, including keyword arguments with default values.

To guard against overly-specific rules, MELT applies *Rule Generalization* (Section IV-C). For example, the template holes :[[x]] and :[y] in the rule in the first row of the bottom of Table I remain unchanged in the match and rewrite templates, indicating that they are not relevant to the change at hand. To enhance the rule's applicability, MELT generalizes the specific argument combination, resulting in an updated version of the rule (shown in the last row of Table I). The revised rule uses a more permissive match template using :[args], which can match any number of function arguments.

III. Mining Pull Requests

In this section, we describe MELT's approach to identify and create code examples for rule inference.

A. Extracting Code Examples from Code Diffs

MELT's input is a pull request $\mathcal{P}$, which contains both natural language descriptions and a set of code diffs $\mathcal{P}.diffs$, each of which corresponds to changed code snippets. However, not all diffs in a pull request are relevant to an API change, as they may encompass unrelated refactoring actions. Therefore, MELT first identifies which changes in the pull request are relevant to the API of interest.

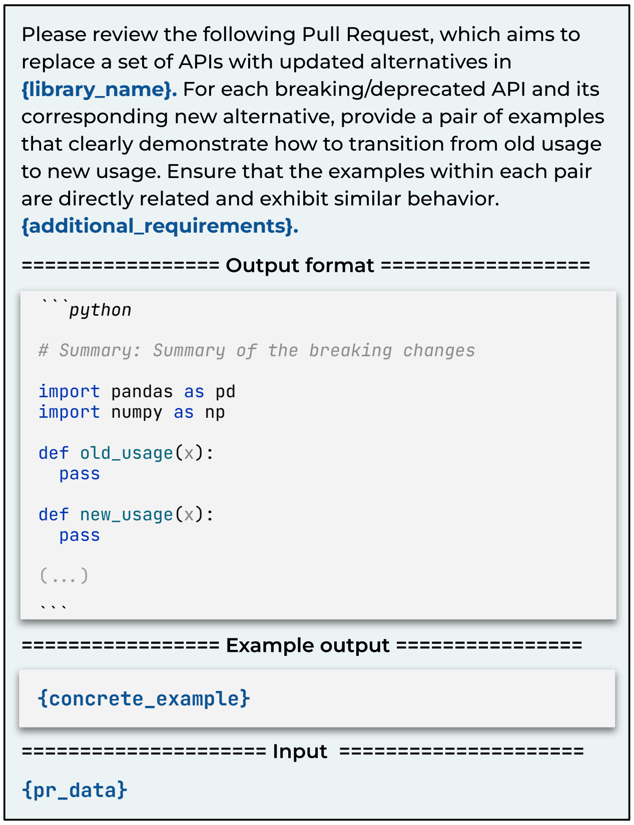
MELT determines which code changes are relevant using its *Code Change Analyzer*. MELT starts by pinpointing which public APIs are affected by the pull request by examining the scope of each code diff to identify the affected function and its corresponding class. For example, for the code change in Figure 1, MELT identifies the function name `test_datatimeindex` and the class where the function comes from `TestSeriesFormatting`. MELT filters out test functions and private namespaces, to exclude API names that are not the main focus of the change.[3] On this example, the test class and method will be filtered, but other changes in the same PR (not shown) affect the `append` and `concat` methods, so MELT considers those methods relevant.

MELT then filters the code diffs to retain only those diffs and surrounding code that contain at least one of the relevant keywords. This produces a set of code examples to serve as inputs to rule inference. For the `pandas` example, although the test method itself is not a relevant API name, the code change in that test method *does* concern relevant API calls, and so these diffs will be retained for use in inference. A strength of this approach is its generalizability across multiple libraries and languages, since it works at token level.

B. Generating Additional Code Examples

As illustrated in Section II, pull requests sometimes lack sufficient code examples to infer migration rules. In a preliminary study, we analyzed 174 pull requests related to breaking changes and deprecations from `pandas`' release notes. We discovered that only 41 (23.6%) of these pull requests contained at least one meaningful code example showcasing the transition from old to new usage. However, pull requests offer other information sources about API changes, including natural language descriptions in comments, developer discussions, and documentation. Our key insight is that this additional data can also be leveraged to generate and test more code examples. MELT uses a *Code Generation Model* to produce extra code examples from this data. Generating code examples rather than the rules directly is advantageous, because we can test and validate the generated code, enhancing confidence in the rules inferred from it. Additionally, the code examples may enhance interpretability by demonstrating the provenance of inferred rules to MELT users.

Algorithm 1 outlines our approach. Given a pull request and a code generation model, MELT iterates for a fixed number of times N (based on the desired number of samples) and asks the model to generate a transition example (line 3). Our GENERATEEXAMPLE implementation prompts `GPT-4 8K` [32], which is well-versed in our target libraries' code, to process PR information (code diffs, title, description, discussion) and generate transition examples for the APIs affected in that PR. Figure 5 shows the template used for the prompt.[4] MELT uses the model to generate a pair of code examples, e_{old} and e_{new}, representing the old and new usages, respectively. While e_{old} uses the old API, e_{new} is implemented using the new API. Both examples are functions with identical signatures but different implementations.



Please review the following Pull Request, which aims to replace a set of APIs with updated alternatives in **{library_name}.** For each breaking/deprecated API and its corresponding new alternative, provide a pair of examples that clearly demonstrate how to transition from old usage to new usage. Ensure that the examples within each pair are directly related and exhibit similar behavior. **{additional_requirements}.**

================= Output format ==================

~~~
```python

# Summary: Summary of the breaking changes

import pandas as pd
import numpy as np

def old_usage(x):
  pass

def new_usage(x):
  pass

(...)
```
~~~

================= Example output ================

`{concrete_example}`

==================== Input ====================

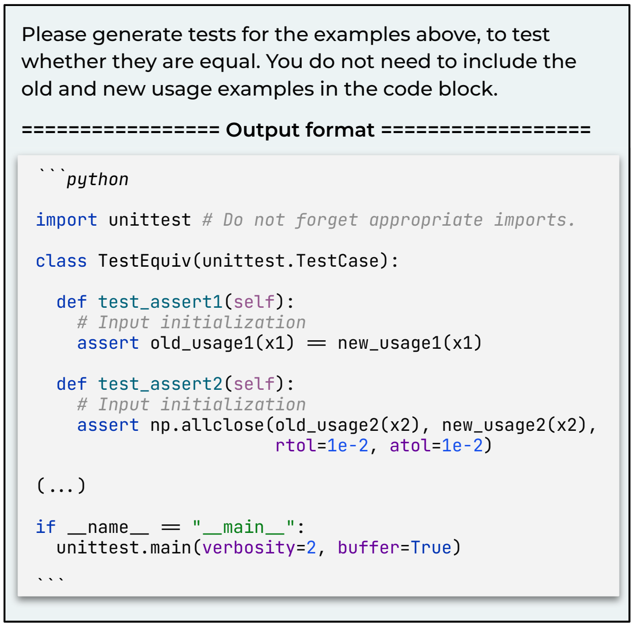
`{pr_data}`

Fig. 5: Prompt template for the GENERATEEXAMPLE function in Algorithm 1, featuring four placeholders: (1) `library_name`, (2) `additional_requirements` for format consistency and correctness, (3) a `concrete_example` with summary and examples from `pandas`, and (4) `pr_data`, the PR information including title, description, changed files, and corresponding diffs, as JSON.

However, simply asking the model to generate a code example is not enough, as there are no guarantees that e_{new} has the same semantics of e_{old}. As a subsequent step, MELT generates test cases that assess the equivalence between e_{old} and e_{new} (line 4). In our implementation of GENERATETESTCASES, MELT follows up with `GPT-4` for test generation. The request includes the original prompt, the model's response, and the text from Figure 6. `GPT-4` generates test inputs and computes

[3] Although our experiments do not exercise this setting, developers can also provide the names of affected APIs when submitting the pull request, which MELT can use directly to eliminate irrelevant code changes.

[4] Full prompts are provided in the artifact at Zenodo [34].



Please generate tests for the examples above, to test whether they are equal. You do not need to include the old and new usage examples in the code block.

================= Output format ==================

~~~
```python
import unittest # Do not forget appropriate imports.

class TestEquiv(unittest.TestCase):

  def test_assert1(self):
    # Input initialization
    assert old_usage1(x1) == new_usage1(x1)

  def test_assert2(self):
    # Input initialization
    assert np.allclose(old_usage2(x2), new_usage2(x2),
                       rtol=1e-2, atol=1e-2)

(...)

if __name__ == "__main__":
  unittest.main(verbosity=2, buffer=True)
```
~~~

Fig. 6: Test case generation prompt. MELT concatenates the prompt from Figure 5, the model's response, and this prompt to ask the model for tests for the generated examples.

```
Algorithm 1 GENERATETRANSITIONEXAMPLES($P, \mathcal{M}, N$)
Input: $\mathcal{P}$: pull request, $\mathcal{M}$: gen model, $N$: number of samples
Output: $E$: transition examples
 1: $E \leftarrow \emptyset$
 2: for $i = 1$ to $N$ do
 3:     $(e_{old}, e_{new}) \leftarrow$ GENERATEEXAMPLE($\mathcal{M}, \mathcal{P}$)
 4:     $\mathcal{T}_E \leftarrow$ GENERATETESTCASES($\mathcal{M}, \mathcal{P}, e_{old}, e_{new}$)
 5:     $E \leftarrow E \cup \{(e_{old}, e_{new})\}$
 6:     for each test $\in \mathcal{T}_E$ do
 7:         if FAILS(test) then
 8:             $E \leftarrow E \setminus \{(e_{old}, e_{new})\}$
 9:         end if
10:     end for
11: end for
12: return $E$
```

their output on e_{old}, which serve as an oracle to test e_{new}. The test case asserts that e_{new} produces the same output as e_{old} for the same set of inputs.

After generating test cases, MELT checks whether any test fails (lines 6-10). If any test case fails, the transition examples for that test are discarded (line 8), as the new usage does not behave similarly to the old one. MELT only considers examples for which test cases were generated. This procedure outputs a set of transition examples (when possible) that can then be used to infer migration and test rules.

IV. RULE GENERATION

MELT uses the Comby language [23] and toolset for searching and refactoring source code [24] to express migration rules. We introduced some elements of the language in Section II, with examples of Comby's syntax-driven match and rewrite templates. Formally, a rewrite rule in Comby is of the form M → R where c$_1$, c$_2$, ..., c$_n$, where M is the match template, R is the rewrite template, and c$_1$, c$_2$, ..., c$_n$ are constraints in the rule language. The key structure of Comby rules are template variables, which are holes in the match and rewrite templates that can be filled with code. Template variable types include, e.g., :[[x]] matching alphanumeric characters (similar to \w+ in regex), and :[x] matching anything between delimiters (e.g., [],(),{}). Comby also supports a small rule language to add additional constraints, like types or regular expression matches, on the template variables. Comby's website [24] provides the full syntax reference. Although language agnostic, Comby is still language *aware*, and can deal with comments and other language-specific constructs. Its rules are also close to the underlying source, and thus typically easier to read than, e.g., transformations over abstract syntax trees.

(a) Code before migration

```
r = pd.read_csv(
  filename,
  compression=comp,
  encoding=enc,
  index_col=0,
- squeeze=True)
```

(b) Code after migration

```
r = pd.read_csv(
  filename,
  compression=comp,
  encoding=enc,
  index_col=0).
+ squeeze()
```

Fig. 7: Example code change from PR #43242 [36] in pandas

The rest of this section describes rule inference.

A. Rule Inference

Given a set of code examples, MELT infers a set of Comby rules that can be used to automatically migrate APIs in client code. First, MELT parses the code files corresponding to each code diff into an abstract syntax tree (AST), identifying the nodes corresponding to the change before and after. MELT then uses a variation of InferRules's algorithm [35] (adapted to Python) that always returns a single rule, and never abstracts away class names, method names, and keyword arguments.

To illustrate, consider the code change in Figure 7, where a library maintainer transforms a keyword argument into a function call. The smallest unit MELT considers for a Comby rule is a source code line. Given the two assignment nodes corresponding to the change, rule inference then abstracts away child nodes with template variables. When a construct has the same character representation, MELT uses the same template variable. For the example, MELT abstracts the left-hand side and right-hand side of both assignments, yielding: :[[a]] = :[b], and :[[a]] = :[c].

Notice that the template variable for the target of both assignments is the same, :[[a]], because their source representation is the same. However, MELT cannot match the right-hand side of the assignments (:[[b]], and :[[c]]). It, therefore further decomposes the AST nodes' children:

```
:[[a]] =                          :[[a]] =
  :[[i]].read_csv(                  :[[i]].read_csv(
  [[d]],                            :[[d]],
  compression=:[e],                 compression=:[e],
  encoding=:[f],            →       encoding=:[f],
  index_col=:[[g]],                 index_col=:[[g]])
  squeeze=:[[h]])                 squeeze()
```

Melt never abstracts away class names, function names, and keyword arguments, as preserving these details is crucial for API migration. Additionally, Melt consistently yields a single, all-encompassing rule. In this case, Melt can match every template variable in the match template with a corresponding node in the rewrite template except `:[[h]]`. Consequently, it attempts to further decompose the nodes, but still fails to match `:[[h]]`, ultimately reverting it and generating the final rule:

```
:[[a]] =                          :[[a]] =
  :[[i]].read_csv(                  :[[i]].read_csv(
  [[d]],                            :[[d]],
  compression=:[e],                 compression=:[e],
  encoding=:[f],            →       encoding=:[f],
  index_col=:[[g]],                 index_col=:[[g]])
  squeeze=True)                   squeeze()

where :[[h]].type == int,
  :[[i]].type == pandas
```

After inferring a rule, Melt incorporates type guards. The goal is to constrain each template hole to its respective observed type. This step is crucial in preventing the misapplication of rules for common API names (e.g., matching `List.append` when the rule targets `DataFrame.append`). In contrast to previous rule synthesis approaches [35], [37], Melt directly incorporates type constraints into `Comby`'s rule language. This integration is possible because we extend `Comby` to support Language Server Protocol (LSP) type inference. Melt uses the `Jedi` [38] type inference language server, making it available for client usage.

B. Rule Filtering

Occasionally, Melt infers spurious rules (e.g., rules that contain variables in the rewrite template that might not be in scope). First, Melt discards *duplicate rules* within the same pull request (post generalization, as well). A rule is considered a duplicate if all of the match, rewrite template and template variable constraints are the same. Melt then further filters by:

1) API Keywords: Melt discards transformation rules that do not contain the name of any affected APIs. This can occur when a developer modifies the surrounding context of a code block, for example, by wrapping a statement in a try-catch block (e.g., `:[x]` → `try:\n:[x]`). These rules are considered spurious because they can match arbitrary code and are not specific to API migration.

2) Unsafe Variable and Private Namespaces: Melt discards rules where a rewrite template uses either variables from private namespaces (indicated by calls with underscores, Python's convention for private attributes/functions/namespaces), or variables not present in the match template. This ensures that the rules do not rely on private or internal functionality that is not accessible to client code.

C. Generalizing Rules

Rules inferred from single code examples may be too specific, as demonstrated in our rule for the `squeeze` example so far. This change is specific to a particular argument combination. However, the `read_csv` function has numerous optional arguments, and the rule should therefore be versatile. Moreover, it can only be applied to assignments, even though the migration applies to other contexts.

Therefore, our approach generalizes rules for broader applicability by abstracting irrelevant context and generalizing arguments. Algorithm 2 overviews the process. Melt obtains AST nodes corresponding to the match and rewrite templates (lines 1-2), and isolates and eliminates all constructs unrelated to the actual code transformation (line 3). Specifically, removeCommonContext unwraps return statements, removes targets on assignments (when possible), and unwraps conditionals, asserts, and other statements, provided they are identical in both the match and rewrite templates. If there are multiple ways to unwrap a statement (e.g., the rule is comprised of two assignments statements), Melt returns the first possible unwrapping.

Next, API call arguments are generalized wherever possible. Melt uses matchings obtained from the Hungarian algorithm during the rule inference process (further explained in [35]) to find matchings between call nodes. Melt examines matching call nodes and generalizes common arguments (line 5). The generalizeArguments function operates by examining pairs of arguments and keyword arguments. If there are multiple consecutive arguments between the match and rewrite templates, we replace the arguments with a generic template variable `:[args]`. Once the arguments of the call pair have been generalized, Melt replaces it in the original templates (lines 6-7). Melt also ensures that keyword arguments always appear at the end of the rewrite template. For example, when a developer turns a positional argument into a keyword argument, the rewrite template moves the positional argument to the position of the last keyword argument. For our running example, the final rule is:

```
:[[i]].read_csv(               :[[i]].read_csv(
  :[args],               →       :[args])
  squeeze=True)                squeeze()

where :[[i]].type == pandas
```

. . . where Melt removed the assignment target and abstracted irrelevant arguments.

Generalization is crucial to ensuring broader rule applicability. However, over-generalization does occur, especially when type information is lost. As a result, generalized rules may need extra validation. However, Melt does allow users to generate rule variations to explore

Algorithm 2 GENERALIZE(r)

```
Input: $r$: a rewrite rule
Output: generalized rewrite rule
$n_1 \leftarrow$ GETBEFORENODE($r$)
$n_2 \leftarrow$ GETAFTERNODE($r$)
$n_1, n_2 \leftarrow$ REMOVECOMMONCONTEXT($n_1, n_2$)
for $(c_1, c_2) \in$ GETCALLPAIRS($n_1, n_2$) do
    ${c_1}', {c_2}' \leftarrow$ GENERALIZEARGUMENTS($c_1, c_2$)
    $n_1 \leftarrow$ REPLACENODE($n_1, c_1, c_1'$)
    $n_2 \leftarrow$ REPLACENODE($n_2, c_2, c_2'$)
end for
return CREATERULE($n_1, n_2$)
```

alternative generalizations, such as a rule with a match template `:[[i]].read_csv(squeeze=True)` or another with more arguments after squeeze `:[[i]].read_csv(:[args0], squeeze=True, :[args1])`. We leave a detailed investigation of these concerns to future work.

V. EVALUATION

We answer the following research questions:

- **RQ1.** How effectively can MELT generate transformation rules from code examples in pull requests?
- **RQ2.** How do code examples generated automatically complement code examples in pull requests?
- **RQ3.** What is the impact of rule generalizability?
- **RQ4.** Are the rules effective for updating client code?

A. Experimental Setup

1) Implementation: Although our approach is largely language-agnostic, we implement it for Python libraries because: (1) Python is one of the most popular programming languages [39], and (2) there exists a gap in migration tools for Python [20]. We implemented rule inference using the Python abstract syntax tree (AST) module. `InferRules` [35] was originally implemented for Java AST; we brought native implementation to Python. We also perform rule generalization at the Python AST level. For code generation, we used the state-of-the-art `GPT-4` [32]. We extended `Comby` to support Language Server Protocol (LSP)-based type inference over match templates [40] with `Jedi` [38], a state-of-the-art static analysis tool. MELT's source code, data, and logs used for the evaluation are available at Zenodo [34].

2) Methodology: We evaluated MELT using four of the most popular Python data science libraries: `numpy`, `scipy`, `sklearn`, and `pandas`. We collected a total of 722 pull requests for `pandas`, 141 for `sklearn`, 186 for `numpy`, and 130 for `scipy` using the `GitHub QL` API and web crawlers over release notes. We took a convenience sampling approach to find PRs concerning API or breaking changes, or deprecation-related PRs, moving backwards from the version of each library (as of April 2023); this includes merged PRs intended for future library releases, as well as those that have been released. We collected more PRs for `pandas` than other libraries because it had a higher number of pull requests, and breaking changes in `pandas` are particularly well documented. We then executed MELT on each pull request.

TABLE II: **RQ1.** *Left:* Pull requests per library, with mined rules and correct rules. *Right*: Filtered and generalized rules mined per library, with total and correct counts.

Library	# PRs	PRs with		Mined Rules	
		Mined Rules	Correct Rules	Total	Correct (%)
`pandas`	722	169	102	521	359 (68.9 %)
`scipy`	130	21	11	33	19 (57.6 %)
`numpy`	186	20	10	47	27 (57.4 %)
`sklearn`	141	38	21	82	56 (68.3 %)
Total	1179	248	144	683	461 (67.5 %)

For our manual assessment of rule correctness and relevancy, two authors of this paper manually labeled a set of rules independently. We defined a *rule to be correct* if (1) it correctly reflects the change in the pull request, and (2) it is generally applicable to client code and does not overgeneralize (i.e., it will not produce incorrect migrations even if it matches the correct APIs in some cases). This procedure requires analyzing the pull request discussion, changes, source code, and documentation when necessary. The annotators discussed five representative examples together and then individually labeled 151 unique rules, achieving an inter-rater reliability (IRR) with a Cohen's kappa of 0.84 (almost perfect agreement) [41]. Due to the high agreement, the first author labeled the remaining rules to cover all research questions.

B. RQ1: Mining Rules from Code Examples in PRs

Table II summarizes MELT's rule inference algorithm on 1179 PRs (722 `pandas`, 130 `scipy`, 186 `numpy`, 141 `sklearn`). MELT's ability to extract code examples from pull requests largely depends on the libraries' testing practices. Nonetheless, a significant number of pull requests contain valuable examples for rule extraction. Previous studies [42] found that only 27.1% of migrations in a different set of libraries were potentially fully automatable. MELT generates correct migration rules for 12.2% of analyzed pull requests, indicating room for improvement (further explored in RQ2).

Running MELT's rule inference algorithm to the 1179 PRs results in 5504 rules. After filtering and generalization, we ended up with 683 rules. The right-most columns of Table II show the number of mined rules after generalization and filtering for each library, and their correctness based on manual validation. On 67.5% of the cases, our mined rules are correct and do not overgeneralize. However, on 32.5% of the cases, MELT derived incorrect, non-generally applicable, or over general rules. We observed three primary reasons for incorrect rules: (1) *Code change not generally applicable*, such that the rule cannot capture the context in which it is applicable. For example, in `numpy` PR #9475 [43], the `np.rollaxis` is deprecated in favor of `np.moveaxis`. Migrating from one API to another depends on the actual content of the variables used in the API, as it behaves differently depending

TABLE III: **RQ2.** *Left:* Code examples generated and passing tests per library. *Middle:* Pull requests with mined and correct ("Corr.") rules from generated examples. *Right:* Filtered and generalized rules per library. *Note:* Limited to 50 PRs per library for budgetary reasons.

	Code Examples		PRs with Rules		Mined Rules		
						Correct	
Library	**Total**	**# pass**	**Total**	**Corr.**	**Total**	**Prev**	**New**
`pandas`	285	134	25	19	45	7	30
`scipy`	194	68	15	13	30	4	18
`numpy`	222	114	21	14	46	2	31
`sklearn`	187	63	21	13	35	5	17
Total	888	379	82	59	156	18	96

on the variables' content. Our rule cannot capture this, as it only considers types, not content. (2) *Overgeneralization* of rule arguments. For instance, `pandas` PR #21954 [44] says "*read_table is deprecated. Instead, use pandas.read_csv passing sep='t' if needed.*". However, one of the inferred rules is `read_table(:[args])` ↦ `read_csv(:[args])`, because the algorithm abstracts all arguments based on the code example. and, (3) *Unrelated* changes not caught by filtering.

> MELT generates **461 correct migrations rules** directly **from code examples for 144 (12.2%) out of 1179 pull requests** from four popular data-science libraries.

C. RQ2: Automatically generated code examples

To evaluate the role example generation played in rule inference, we sampled 50 pull requests for each library (limited by budget). We used a template to create a prompt to ask the model to generate both code examples and test cases/inputs for the examples, per pull request. The prompt includes the title, description, discussion, and code changes. We used OpenAI's API to prompt `GPT-4`, with a (default) temperature of 0.2, and sampled the model with $N = 5$ in Algorithm 1.

The left side of Table III shows the number of unique examples generated for each library and the number of examples that passed the test suite. MELT produced 248 unfiltered and ungeneralized rules on these examples; filtering and generalization produced *156 unique rules*. We also assessed whether these rules could have been generated from the pull request code directly, by checking (1) whether they were mined in RQ1 (Section V-B), or (2) whether they could be directly applied to their corresponding pull request (meaning that they *could* have been mined in RQ1, but may have been heuristically filtered away).

Table III summarize rule mining success using generated examples by pull request (middle columns); the right-hand side shows the number of rule mined. We categorized correct rules into those that could have been mined without new examples (prev), and those that are new with the generated examples. Like in the previous RQ, MELT can generate incorrect rules in some scenarios. Consider the following example rule: `:[[aah]].shift(:[aae], fill_value=:[aaf])` → `:[[aah]].shift(:[aae], fill_value=pd.Timestamp(:[aaf]))`. [5] This rule is derived from `pandas` pull request number #49362 [45]. The release notes for the PR state: *"Enforced disallowing passing an integer fill_value to DataFrame.shift and Series.shift with datetime64, timedelta64, or period dtypes"*. This transformation is only valid if the series has a `datetime64` `dtype object`, a condition not captured by the rule. While the transformation correctly preserves behavior in this instance, it is incorrect for general application. More diverse tests for the code example could likely increase coverage and filter more incorrect rules.

> MELT generated **114 correct rules** out of the 156 generated rules (73.1%) from **auto-generated transition examples**. **96 (61.5%) of those rules would not have been generated otherwise**.

D. RQ3: Generalizability

Of the 156 rules we manually validated in RQ2, 41 had generalized arguments, and only 9 (22%) were incorrect. To further evaluate the impact of generalizability with an ablation study, by disabling the generalization procedure. We selected 15 rules that had been generalized, along with their non-generalized counterparts. Using Sourcegraph's code search [46],[6] we searched for repositories containing a given keyword in the rule (e.g., for `readcsv(..., squeeze=True)`, we searched for `squeeze=True`). We then cloned up to 50 random repositories for each rule, and ran the generalized and non-generalized rules on these repositories, counting matches.

Table IV shows matches for original and generalized rules, showing that generalization significantly improves rules applicability. For instance, the number of matches for the `set_index` case increased from 2 to 370 (185x) with generalization. Generalization is important because it abstracts context unrelated to API changes. As we focus on API migration in Python, where there can be many argument combinations (e.g., APIs with as many as 10 keyword arguments), generalization helps capture the essence of the change by abstracting arguments. Some rules had 0 matches because `Comby` was unable to infer types (`Comby` does not apply rules when it cannot infer types of a template match), or the query was poorly constructed.

> Generalization led to a **9.07x increase** in rule matches, boosting potential rule applications from **162** to **1469** in our sample. This demonstrates the **significant impact of generalization on rule applicability**.

E. RQ4: Updating client code

To evaluate the effectiveness of our approach to updating developer code, we migrated outdated library API usage in developer projects found on GitHub for the `sklearn`, `pandas`, and `scipy` libraries. Collecting and running client projects

[5] Template variables are omitted for brevity.

[6] Note SourceGraph only indexes repositories with at least two stars.

TABLE IV: **RQ3.** Comparison of Non-General and Generalized Rules

Library	Original Rule		Generalized Rule	
	Match Template	**Matches**	**Match Template**	**Matches**
pandas	`:[[x]].set_index(:[a], drop=:[[b]], inplace=True)`	2	`:[[x]].set_index(:[args], inplace=True)`	370
	`:[[x]].read_csv(:[[a]], compression=:[[b]], encoding=:[[c]], index_col=:[d], squeeze=True)`	0	`:[[x]].read_csv(:[args], squeeze=True)`	21
	`:[[aai]].apply(:[a], axis=:[[b]], reduce=True)`	3	`:[[aai]].apply(:[args], reduce=True)`	4
scipy	`jaccard_similarity_score(:[[a]], :[[b]])`	94	`jaccard_similarity_score(:[args])`	226
	`:[[x]].filters.gaussian_filter(:[a], :[b], mode=:[[c]])`	0	`:[[x]].filters.gaussian_filter(:[args])`	86
	`:[[x]].query(:[[a]], :[[b]], n_jobs=:[c])`	0	`:[[x]].query(:[args], n_jobs=:[y])`	0
	`:[[x]].hanning(:[[a]], :[[b]])`	0	`:[[x]].hanning(:[args])`	0
numpy	`:[[x]].alltrue(:[a], axis=:[b])`	7	`:[[x]].alltrue(:[args])`	208
	`:[[x]].histogram(:[[a]], bins=:[b], range=:[c], normed=:[y])`	2	`:[[x]].histogram(:[args], normed=:[y])`	66
	`:[[x]].complex(:[[a]], :[[b]])`	17	`:[[x]].complex(:[args])`	20
sklearn	`BaggingClassifier(base_estimator=:[[a]], n_estimators=:[[b]], random_state=:[[c]])`	26	`BaggingClassifier(base_estimator=:[x], :[args])`	220
	`BaggingRegressor(base_estimator=:[[a]], n_estimators=:[[b]], random_state=:[[c]])`	7	`BaggingRegressor(base_estimator=:[x], :[args])`	116
	`KMeans(n_clusters=:[a], init=:[[b]], n_init=:[[c]], algorithm='full')`	0	`KMeans(:[args], algorithm='full')`	38
	`AgglomerativeClustering(n_clusters=:[a], linkage=:[b], affinity=:[c])`	4	`AgglomerativeClustering(:[args], affinity=:[c])`	28
	`OneHotEncoder(sparse=:[[aac]], categories=:[[aan]], drop=:[[aaz]])`	0	`OneHotEncoder(sparse=:[x], :[args])`	66

TABLE V: **RQ4.** Effects of rule application on developer projects.

Library	Total Projects	Affected Projects	Unique Rules	Rule Applications	Additional Warnings	Resolved Warnings	Additional Passing Tests	Additional Failures	Resolved Failures
sklearn	20	10	6	27	9	598	2	1	1
pandas	20	10	4	23	0	44	7	81	7
scipy	20	6	5	23	0	266	0	1	0
Total	60	26	15	73	9	908	9	83	8

requires significant manual effort: many projects do not specify dependencies or provide tests. We therefore did not evaluate numpy API usage, but we can expect similar results.

We found client projects by searching GitHub for public repositories that used outdated versions of each library, and included code that matched to at least one of the match templates of an inferred rule from RQs 1 and 2. We applied a total of 15 unique rules across the three libraries. We provide detail on specific rules and projects in Zenodo [34]. For each library, we identified 20 client projects that used outdated versions, and between one and three rules applied. We cloned each project, updated its library dependencies to a version with the breaking change, installed necessary dependencies, and ran all tests to note passing tests, failures, errors, and warnings. We then used Comby to automatically update the outdated API usage, and reran the tests to compare results post-migration. We did this separately for each applicable rule.

Table V summarizes results. Total Projects refers to the total number of projects to which we applied rules and tested. Affected Projects refers to the number of evaluated projects that had a change in the tests after rule application from new or resolved warnings, passed tests, or failures. Not all of the projects had tests affected by rule application, either because test coverage was incomplete or because persistent failing tests in developer projects obscured the effect of rule application.

For sklearn, slightly less than half the developer project tests were affected by rule application. Only two of the projects showed a negative impact of rule application, where one project had an additional failing test and another project had nine new warnings. The sklearn rules were applied without type information, which is one potential cause for the negative impact. The other affected projects had warnings resolved, ranging from 1 to 563 warnings resolved for a single project. One project had additional passing tests.

For pandas, rule application affected half of client projects. While there were 81 additional failures from pandas rules, they were isolated to four projects and a single rule. These new failures occurred because of a lack of type information, meaning one rule was erroneously applied to API calls unrelated to the pandas library. In other projects, the same rule was applied correctly, even without type information, and successfully resolved warnings. The other three unique pandas rules were applied with type information. No pandas rules introduced new warnings.

For scipy, rules were also applied absent type information, but only one application introduced an error. All six affected scipy projects had warnings resolved by rule application, and none of the scipy rule applications caused additional warnings.

Of the 60 evaluation repositories, 34 had no change in the tests or warnings. However, this does not indicate that rule transformation was incorrect or unnecessary: most projects had failing tests and errors unrelated to API usage, which can obscure the effect of rule application. Overall, the resolved warnings and failures demonstrate MELT's potential to help developers more easily maintain large projects.

VI. Discussion

In this section, we address the main limitations of our method and possible future work.

A. Limitations and threats

Rule correctness. We used manual validation to assess rule correctness, with a process that entailed high IRR kappa indicating agreement. One approach for further validation could involve upgrading client projects to newer library versions and applying the rules on projects using these libraries. In RQ4, we use this method to demonstrate that some rules are indeed correct. However, this process is challenging. MELT does not mine rules for *all* breaking changes in a given release, so upgrading client projects may break multiple aspects in ways automatic find-and-replace rules cannot address [42]. However, automating a large part of migration in ways that entail minimal additional technology or effort on the part of the client developer holds promise for reducing the challenge of upgrading library dependencies. Our rules could also potentially be validated using differential testing techniques or by requesting more tests from the code generation model.

Code generation model. Our approach relies on a code generation model to generate examples when none are available. We selected GPT-4, a state-of-the-art model trained on data before September 2021. We successfully evaluated on pull requests opened after September 2021, demonstrating the risk of data leakage in these experiments is low. The model, however, is paid and not open-source. As AI research advances, we anticipate better models being made public. We opt for a model-based code generation approach over generating Comby rules directly because rules can be validated with code examples (if the code does not pass, we discard the example). Additionally, the model is not fine-tuned and has limited exposure to Comby, and is likely to work better on commonly-used languages like Python. For less popular APIs, however, fine-tuned versions of the model on library code might be necessary.

Generalization. Our generalization procedure removes context and arguments that appear unrelated to the change, only considering diffs. Removing too much context and type information may result in spurious rules. Conversely, insufficient generalization can make the rule too specific. MELT can return both rules to the user, allowing them to decide what to keep. Currently, developers must manually validate rules to ensure they make sense. To facilitate this, we developed a CI solution on GitHub for integrating our tool. Rules can be validated and modified, if necessary, by whoever merges the PR, or automatically validated, as previously discussed.

B. Comparison against prior work

Few API migration tools target Python, challenging direct comparison to prior work. MELT adapts its inference algorithm from InferRules [35], designed for type migration in Java. Consequently, MELT without generalization and filtering serves as a baseline equivalent to InferRules. The most closely related approach, PyEvolve [37], builds on InferRules using Comby as an intermediate representation. PyEvolve focuses on general refactoring, and adapts rules to different control variants, requiring more complex analysis, and client code analysis. This is in contrast to MELT's lightweight approach, which aims to minimize overhead on client developers. Since most of our rules are 1:1 and 1:n transformations, adapting rules for control flow variants is less relevant. Overall, while PyEvolve is more powerful in the types of rules it can infer, fundamentally it serves a different goal as compared to MELT.

Our evaluation differs from closely-related prior work [12], [14] in two ways. First, our manual validation process is able to consider more information in the form of the PR and library documentation. That is, rather than looking at rules in isolation or limiting attention to syntactic validity, we can consider whether the change actually reflects PR intent. Second, we provide an end-to-end evaluation of automatically inferred rules on a number of client code repositories, complementing manual rule validation.

As we discuss in Section VII, most prior approaches for automatic API migration (or code evolution generally) mine migration examples from client projects or their source control histories. MELT relies solely on the changed *library*, looking at internal code changes to inform rule mining. This allows MELT to apply earlier in the library update process. However, libraries do not always include sufficient changed code examples to inform migration, which is why MELT also prompts an LLM to generate extra examples, along with tests to validate those examples. Other approaches may also benefit from using LLMs this way, particularly those whose use cases entail fewer available examples, like A3 [16] (focusing on Android API migration), or APIFix [14] (evaluated on changes to library code, similar to MELT). APIFix in particular could likely benefit from the LLM-generated examples and tests, because it uses edit examples in its program synthesis algorithm. Other tools are evaluated across many more example changes to client code, like Meditor [12]. These approaches may not require new examples, but leveraging LLMs may allow them to apply earlier in the update process, or in scenarios where migration examples are scarce. Indeed, as models with larger context windows become available (e.g., CLAUDE 100K token context [47]), it becomes possible to include more comprehensive data in prompts, such as full API documentation. This suggests a promising avenue for generating higher-quality, context-rich examples for rule mining, particularly when extant migration examples are scarce.

VII. Related Work

Empirical studies on API evolution. API evolution has long been a challenging software engineering concern without a definitive solution. Developers often lag in updating their software to the latest APIs, leading to compatibility issues and hindering maintenance [19]. Recent work identifies a significant need for more support for API evolution tools in languages other than Java, particularly Python [20]. MELT aims to address this gap. Moreover, Dilhara et al. [48] further reinforce the need for migration tools for Python, finding that Python data library clients tend to need to update dependencies frequently and face significant challenges in doing so.

A study of API migration in four popular Java libraries found that only 27.1% of these migrations were fully automatable [42]. This suggests that achieving 100% safe and automated migration rules is unlikely, as some transformations are complex and need more context than rule-based approaches can provide. MELT's imperfect accuracy aligns with these findings, as some rules are incorrect simply due to the difficulty in capturing them with purely syntax-driven transformations. However, MELT's approach can provide semi-automated support for migration, easing the overall burden.

Meanwhile, refactoring tools that require developers to use complex domain-specific languages are often difficult to use and, consequently, often poorly-adopted [49]. This observation emphasizes the need to develop user-friendly and ergonomic API migration tools and techniques that seamlessly integrate into developers' workflows, as MELT aims to do.

Library evolution. Automated API migration research has primarily focused on mining client repositories, usually targeting object-oriented languages (namely Java and C#). For example, A3 [16] and Meditor [12] mine client repositories for examples to create rules, which are then customized to new clients. APIFix [14] mines transition examples from both previously-migrated and new client repositories and uses Refazer's [50] engine to learn rules. Refazer's transformations are expressed as AST edits, which are more difficult to understand [51]. Unlike previous approaches, MELT emphasizes simplicity via lightweight find-and-replace transformations.

APIMigrator [15] and AppEvolve [17] also mine client repositories for transition examples and apply them directly to clients. Both tools use differential testing to validate edits on clients. MELT focuses on generating rules rather than updating client code directly, however, it could similarly benefit from incorporating differential testing [52] to validate inferred rules.

Approaches like Semdiff [53] recommends API changes to developers by presenting a ranking of potential replacements.

Code Refactoring. Catchup! [54] records refactorings made by library developers during development and replaces them in client code. LASE [55] and SYEDIT [56] mine code examples for systematic edits, generating edit scripts at the AST level rather than using find-and-replace rules. InferRules [35], [57] inspired our rule inference algorithm. However, InferRules primarily targets type migration and extensively mines client repositories for refactoring examples.

PyEvolve [37], developed concurrently with this work, also uses InferRules's algorithm to infer Comby rules from code changes. However, its purpose is different, as we discussed in detail in Section VI-B. Most migrations are either 1:1 (47.2%) or 1:n (48.1%) [42], and control-flow awareness is not necessary for API evolution. MELT focuses on rule generalization instead. This reduces analysis overhead, especially for large code bases, as only Comby needs to be run.

SOAR [13] uses program synthesis to refactor client code rather than generate find-and-replace rules, aiming to support migration *between* libraries. Unlike MELT, SOAR refactors client code as a blackbox, which can be less interpretable. It can handle more complex migrations, but is commensurately less performant. To the best of our knowledge, PyEvolve and SOAR are the only two tools besides MELT that can infer and apply refactorings for Python code.

VIII. Conclusion

Selecting and maintaining APIs is critical yet challenging in software development. Developers may have to manually update APIs due to evolving libraries, which is time-consuming and error-prone process. We present MELT, which assists developers by generating lightweight API Migration rules in Comby. Unlike previous approaches, MELT mines rules directly from library pull requests instead of client projects. This approach allows rule inference to be integrated directly into the library workflow, eliminating the need to wait for clients to migrate their code. Furthermore, MELT rules are purely syntax-driven, and require no additional tooling on client side (besides Comby). We evaluated MELT on pull requests from four popular libraries: pandas, scipy, numpy, and sklearn. We assessed rule accuracy by examining the pull request descriptions, discussions, and more. We discovered 461 accurate rules from code examples in pull requests and 114 rules from auto-generated code examples. To show practical applicability, we applied the rules to client projects and ran their tests, proving their effectiveness in real-world situations.

Acknowledgements

This work was supported by Portuguese national funds through FCT under projects UIDB/50021/2020, PTDC/CCI-COM/2156/2021, 2022.03537.PTDC and grant SFRH/BD/150688/2020, as well as the US National Science Foundation, Awards CCF-1750116 and CCF-1762363.

References

[1] C. R. De Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "How a good software practice thwarts collaboration: the multiple roles of apis in software development," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 221–230, 2004.

[2] L. Xavier, A. Brito, A. C. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *Proc. IEEE International Conference on Software Analysis, Evolution and Reengineering*, M. Pinzger, G. Bavota, and A. Marcus, Eds., 2017, pp. 138–147.

[3] C. Bogart, C. Kästner, J. D. Herbsleb, and F. Thung, "How to break an API: cost negotiation and community values in three software ecosystems," in *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 109–120.

[4] D. Dig and R. E. Johnson, "How do apis evolve? A story of refactoring," *Journal of Software Maintenance: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[5] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[6] J. H. Perkins, "Automatically generating refactorings to support API evolution," in *Proc. ACM Workshop on Program Analysis for Software Tools and Engineering*, 2005, pp. 111–114.

[7] M. Fowler, "Refactoring: Improving the design of existing code," in *XP Universe and First Agile Universe Conference*, ser. Lecture Notes in Computer Science, D. Wells and L. A. Williams, Eds., vol. 2418. Springer, 2002, p. 256.

[8] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, p. 50.

[9] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "Data scientists in software teams: state of the art and challenges," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds., 2018, p. 585.

[10] M. Dilhara, A. Ketkar, and D. Dig, "Understanding software-2.0: A study of machine learning library usage and evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 55:1–55:42, 2021.

[11] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: counting those that matter," in *Proc. ACM International Symposium on Empirical Software Engineering and Measurement (ESEM)*, M. Oivo, D. M. Fernández, and A. Mockus, Eds., 2018, pp. 42:1–42:10.

[12] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of API migration edits," in *Proc. of the International Conference on Program Comprehension*, Y. Guéhéneuc, F. Khomh, and F. Sarro, Eds. IEEE / ACM, 2019, pp. 335–346.

[13] A. Ni, D. Ramos, A. Z. H. Yang, I. Lynce, V. M. Manquinho, R. Martins, and C. Le Goues, "SOAR: A synthesis approach for data science API refactoring," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 112–124.

[14] X. Gao, A. Radhakrishna, G. Soares, R. Shariffdeen, S. Gulwani, and A. Roychoudhury, "Apifix: Output-oriented program synthesis for combating breaking changes in libraries," in *Proc. ACM SIGPLAN Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, vol. 5, 2021, pp. 1–27.

[15] M. Fazzini, Q. Xin, and A. Orso, "Apimigrator: an api-usage migration tool for android apps," in *Proc. IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, D. Lo, L. Mariani, and A. Mesbah, Eds., 2020, pp. 77–80.

[16] M. Lamothe, W. Shang, and T. P. Chen, "A3: assisting android API migrations using code examples," *IEEE Transactions on Software Engineering (TSE)*, pp. 417–431, 2022.

[17] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *Proc. of the International Symposium on Software Testing and Analysis*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 204–215.

[18] R. G. Kula, D. M. Germán, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration," *Springer Empirical Software Engineering (ESE)*, vol. 23, no. 1, pp. 384–417, 2018.

[19] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proc. IEEE International Conference on Software Maintenance*, 2013, pp. 70–79.

[20] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of api evolution literature," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–36, 2021.

[21] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 345–355.

[22] Y. Yu, H. Wang, V. Filkov, P. T. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on github," in *Proc. ACM IEEE International Conference on Mining Software Repositories (MSR)*, M. D. Penta, M. Pinzger, and R. Robbes, Eds. IEEE Computer Society, 2015, pp. 367–371.

[23] R. van Tonder and C. Le Goues, "Lightweight multi-language syntax transformation with parser parser combinators," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, pp. 363–378.

[24] R. van Tonder. (2022, Jul.) Comby. [Online]. Available: https://comby.dev/docs/overview

[25] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of API migration edits," in *Proc. ACM IEEE International Conference on Program Comprehension (ICPC)*, Y. Guéhéneuc, F. Khomh, and F. Sarro, Eds., 2019, pp. 335–346.

[26] M. Fazzini, Q. Xin, and A. Orso, "Apimigrator: an api-usage migration tool for android apps," in *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, D. Lo, L. Mariani, and A. Mesbah, Eds. ACM, 2020, pp. 77–80.

[27] M. Lamothe, W. Shang, and T. P. Chen, "A3: assisting android API migrations using code examples," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 2, pp. 417–431, 2022.

[28] A. Ketkar, O. Smirnov, N. Tsantalis, D. Dig, and T. Bryksin, "Inferring and applying type changes," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*. ACM, 2022, pp. 1206–1218.

[29] J. Reback, "DEPR: Deprecate DataFrame.append and Series.append," https://github.com/pandas-dev/pandas/pull/44539, 2022, [Online; accessed May 02, 2023].

[30] pandas-dev, "pandas-dev/pandas: Powerful data structures for data analysis, time series, and statistics," https://github.com/pandas-dev/pandas, 2022, [Online; accessed May 02, 2023].

[31] "Scipy: Open source scientific tools for python," https://www.scipy.org/, 2023, accessed: May 2, 2023.

[32] OpenAI, "GPT-4," https://openai.com/research/gpt-4/, accessed: May 2, 2023.

[33] "deprecate signal.spline in favor of signal," https://github.com/scipy/scipy/pull/14419, accessed: May 2, 2023.

[34] D. Ramos, "Replication of MELT: Mining Effective Lightweight Transformations from Pull Requests," https://doi.org/10.5281/zenodo.8226234, 2023.

[35] A. Ketkar, O. Smirnov, N. Tsantalis, D. Dig, and T. Bryksin, "Inferring and applying type changes," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*. ACM, 2022, pp. 1206–1218.

[36] "DEPR: squeeze() argument in read_csv/read_table #43242," GitHub Pull Request Pandas #43242, accessed on May 5, 2023 https://github.com/pandas-dev/pandas/issues/43242.

[37] M. Dilhara, D. Dig, and A. Ketkar, "Pyevolve: Automating frequent code changes in python ml systems," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, 2023.

[38] D. Halter, "Jedi: An autocompletion tool for python," https://jedi.readthedocs.io/en/latest/, accessed: May 2, 2023.

[39] T. S. BV, "Tiobe index," https://www.tiobe.com/tiobe-index/, 2023, accessed on May 3, 2023.

[40] R. van Tonder, "Comby with types," https://comby.dev/blog/2022/08/31/comby-with-types, August 2022, accessed on 3 May 2023.

[41] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[42] B. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, W. Tracz, M. P. Robillard, and T. Bultan, Eds. ACM, 2012, p. 55.

[43] "Dep: deprecate rollaxis," GitHub Pull Request Numpy #9475, accessed on May 4, 2023 https://github.com/numpy/numpy/pull/9475.

[44] "DEPR: pd.read_table," GitHub Pull Request Pandas #21954, accessed on May 4, 2023 https://github.com/pandas-dev/pandas/pull/21954.

[45] "DEPR: disallow int fill_value in shift with dt64/td64 #49362," GitHub Pull Request Pandas #49362, accessed on May 5, 2023 https://github.com/pandas-dev/pandas/issues/49362.

[46] "Sourcegraph," https://sourcegraph.com/, accessed on May 5, 2023.

[47] Claude, "100k context windows," Anthropic, 2023, accessed on July 24, 2023, https://www.anthropic.com/index/100k-context-windows.

[48] M. Dilhara, A. Ketkar, and D. Dig, "Understanding software-2.0: a study of machine learning library usage and evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–42, 2021.

[49] J. Kim, D. S. Batory, and D. Dig, "Scripting parametric refactorings in java to retrofit design patterns," in *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME)*, R. Koschke, J. Krinke, and M. P. Robillard, Eds., 2015, pp. 211–220.

[50] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 404–415.

[51] W. Ni, J. Sunshine, V. Le, S. Gulwani, and T. Barik, "recode : A lightweight find-and-replace interaction in the IDE for transforming code by example," in *Proc. ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, J. Nichols, R. Kumar, and M. Nebeling, Eds. ACM, 2021, pp. 258–269.

[52] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on security and privacy (SP)*, 2017, pp. 615–632.

[53] B. Dagenais and M. P. Robillard, "SemDiff: Analysis and recommendation support for API evolution," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, 2009, pp. 599–602.

[54] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, G. Roman, W. G. Griswold, and B. Nuseibeh, Eds., 2005, pp. 274–283.

[55] N. Meng, M. Kim, and K. S. McKinley, "LASE: locating and applying systematic edits by learning from examples," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., 2013, pp. 502–511.

[56] N. Meng, M. Kim, and K. McKinley, "Systematic editing: generating program transformations from an example," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 329–342.

[57] O. Smirnov, A. Ketkar, T. Bryksin, N. Tsantalis, and D. Dig, "Intellitc: Automating type changes in intellij IDEA," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, 2022, pp. 115–119.