# Automated API Refactoring for Evolving Codebases

**Daniel Rosa Ramos**

CMU-S3D-25-113
August 2025

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**

Claire Le Goues, Chair (Carnegie Mellon University)
Ruben Martins (Carnegie Mellon University)
Joshua Sunshine (Carnegie Mellon University)
Nuno Lopes (Instituto Superior Técnico)
Vasco Manquinho (Instituto Superior Técnico)
Işıl Dillig (University of Texas, Austin)

*Submitted in partial fulfillment of the requirements*
*for the Degree of Doctor of Philosophy in Software Engineering*

© 2025 Daniel Rosa Ramos

This work was supported by Portuguese national funds through the Fundação para a Ciência e a Tecnologia, through the CMU Portugal Program, under the grant SFRH/BD/150688/2020.

**Keywords:** API refactoring, code migration, large language models, program synthesis

# Abstract

Modern software development depends heavily on third-party libraries and frameworks, which expose their functionality through APIs and bring substantial productivity gains. However, as libraries evolve to meet new technical or market demands, clients must often adapt their code to accommodate breaking changes or even newer libraries. This form of software maintenance, known as API refactoring, is a time-consuming and error-prone task, which has led to significant interest in automating it. A common approach to automating API refactoring is to mine historical data from client repositories to extract match-replace rules. However, these approaches are limited by the availability of high-quality examples: many clients do not refactor in public, and those that do leave insufficient traces to learn from.

This thesis presents a set of alternative methods for learning API migration rules without requiring large-scale mining of client code. Instead, we explore three complementary sources of information: documentation, the API development process, and natural language. First, we use API documentation to infer mappings between old and new APIs, which guide the synthesis of migration scripts. Second, we extract migration knowledge from the evolution of the library itself, especially from pull requests that introduce breaking changes and update internal tests. Finally, we show that large language models trained on natural language artifacts can be used to generate migration examples, which are then validated and generalized into reusable scripts. We operationalize these ideas in four refactoring tools, each targeting a different aspect of the problem. These tools combine program synthesis with machine learning to synthesize and apply migrations automatically. We evaluated our techniques in real-world Python libraries and synthetic benchmarks, showing that it is possible to automate migration effectively using only indirect sources of information, without requiring curated datasets or repository mining.

# Acknowledgments

I would like to begin by expressing my deepest gratitude to my advisors, Claire Le Goues, Ruben Martins, Vasco Manquinho, and Inês Lynce. A PhD is the culmination of many years of work and countless interactions, and I have been fortunate to share this journey with each of you.

I also wish to thank my committee members, Nuno Lopes, Joshua Sunshine, and Isil Dillig, for taking the time to read and discuss my work. This dissertation would not have been as strong without your helpful and thoughtful feedback.

This work has also been shaped by the incredible people I have had the privilege of working alongside in my two research groups. I am especially thankful to everyone in *squaresLab* for the discussions, advice, and camaraderie that enriched my experience: He Ye, Aidan Yang, Cláudia Mamede, Harrison Green, Kush Jain, Luke Dramko, Nikitha Rao, Trenton Tabor, Tobias Dürschmid, Paulo Santos, Chris Timperley, Jeremy Lacomis, Leo Chen, and Sophia Kolak.

I am equally grateful to my colleagues in the *ARSR* group at INESC-ID for their friendship and support throughout these years: Pedro Orvalho, Ricardo Brancas, Margarida Ferreira, Carolina Carreira, Henrique Guerra, Nuno Saavedra, João Cortes, Martim Afonso, and Bruno Lourenço.

I would also like to thank the many other people I met along the way: Catarina Gamboa, Maria Casimiro, Hugo Simão, Nadia Nahar, Parv Kapoor, Luís Gomes, Vasu Vikram, Jenny Liang, Kyle Liang, Manisha Mukherjee, Chenyang Yang, and many others whose friendship and company made these years far more rewarding.

I am grateful to all the CMU and IST faculty from whom I learned so much. In particular, I wanted to thank Justine Sherry for being so welcoming and kind when I first arrived in the USA.

I am grateful to the S3D staff at CMU, including Jennifer Cooper, Connie Herold, and Alisha Roudebush, and to the INESC-ID staff: Teresa Mimoso, Vanda Fidalgo, and Paula Barrancos, for their support.

I was also fortunate to gain first-hand experience in software engineering practice during two summers with the Programming Systems Group at Uber. I am thankful to Raj Barik, Ameya Ketkar, Lázaro Clapp, Yuxin Wang, Stefan Heule, Sonal Mahajan, Milind Chabbi, Diego Marcilio, and the many others I met there.

A special word of thanks goes to my oldest friends, Cláudia Alves and Filipa Bernardino, for the many good times we have shared over the years. Though our paths have been very different, your friendship has been constant and invaluable. I am also deeply grateful to all my other friends (near and far, old and new) whose support, laughter, and encouragement have helped me along the way.

Finally, to my family, my parents, my brother, and my grandparents, for their endless support. And to you, whoever is reading this, and to all those whose names are not listed here but have been part of this journey: thank you.

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

## Contents



## 1.1 Motivation

Modern software development relies heavily on third-party libraries and frameworks. These libraries facilitate software reuse [1], allow developers to leverage quality implementations for a desired functionality, and yield significant productivity benefits [2]. Libraries expose their functionality through *Application Programming Interfaces (APIs)*. APIs serve as contracts between the library developers and its clients [3], providing functionality through a set of functions and methods, and hiding concrete implementation details [4, 5]. Although stable API selection is desirable, the dynamic nature of software often renders it impractical. This dynamism in software [6] is driven by changing technical requirements and shifts in stakeholder or market needs [7]. As requirements and libraries evolve, clients may need to migrate APIs to accommodate these shifts [7]. Further-

more, APIs themselves might break, become deprecated [8], or become exploitable, posing severe security risks, thus forcing downstream clients to update their usage accordingly.

The task of changing APIs to accommodate non-functional changes is a specific instance of *software refactoring*, a crucial software engineering practice. Refactoring involves modifying code with the goal of enhancing its quality and reducing its overall complexity [9]. Broadly, refactoring facilitates new feature development [10], assists managing technical debt [11], and prevents codebase decay [12]. Neglecting to refactor can escalate future costs; for example, the Consortium for Information & Software Quality (CISQ) estimates the cost to address the accumulated technical debt in the U.S. at approximately $1.31 trillion [13].

However, refactoring is generally labor-intensive and error-prone [14]. Even seemingly straightforward tasks, like moving between two libraries that provide similar functionality, can be challenging and tricky. For instance, *PyTorch*[15] and *Tensorflow*[16], two of the most popular deep learning libraries, have different conventions regarding the order of values in tensor dimensions, which can lead to subtle bugs during a migration. Migrating APIs requires significant domain-specific expertise in both the source and target libraries [17]. Furthermore, achieving the desired functionality often extends beyond simple method mappings. Developers may need to write additional boilerplate code, figure out the correct argument combination in the replacement API, and manage cascading changes like type migrations and removing/adding import statements. Migration tasks are also difficult as APIs continuously evolve, often rendering prior knowledge obsolete.

The complexity of API refactoring has inspired numerous research efforts towards the automation of this task. At a high level, the goal is to automatically infer and generalize API refactorings from minimal user/developer input in order to reapply them on a large scale. These refactorings include migrations, updates from breaking changes, or handling deprecations. Like in any automation task, the goal is to orchestrate refactoring in such a way that user intervention is minimal. However, full automation is hard, and existing tools typically provide partial support rather than a fully integrated refactoring experience (e.g., [18–22]).

The most common approach to automated API refactoring is to use heuristics or learning algorithms to establish mappings between APIs and create match-replace rules [23]. The data for these algorithms is typically sourced from large-scale collaborative coding platforms like GitHub [24]. The data is obtained either by mining commits from library client projects that have undergone migrations [19, 21, 22], and can also be supplemented by information from new client projects in the most up-to-date APIs [18].

One significant challenge with these mining approaches is that the effectiveness of the tools is limited by their reliance on mining data from client projects that have already refactored their APIs. Unfortunately, this data is scarce. For example, a recent study found that 81.5% of projects maintain outdated dependencies [25]. 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 [26, 27]. On the other hand, unsupervised learning methods [28] circumvent the issue of pair-wise data scarcity. However, they require extensive training data. This would in theory primarily benefit well-established libraries and APIs but, in practice unsupervised

methods are not as effective. Moreover, they cannot tackle lesser-known libraries and proprietary code.

## 1.2 Thesis

In this work, we present novel methods for automated API refactoring that require neither extensive training data nor pairwise migration examples from downstream client projects. Specifically, we tackle two refactoring challenges: 1. **cross-library migrations**, and 2. **same-library migration to cope with breaking changes**. Our techniques are based on two different approaches:

1. inferring match-replace rules for updating APIs;
2. refactoring the APIs directly using a tool.

We observe that a wealth of high-quality information for automated API refactoring can be sourced from API documentation, the development processes of the APIs, and other self-contained information in libraries, along with natural language associated with the API development process.

**Thesis Statement**

*Self-contained data in libraries and synthesis techniques enable automated API refactoring, by supporting the generation of migration scripts and facilitating code migration without relying on pairwise training data.*

Next, we explain the hypotheses behind the thesis statement.

### 1.2.1 API Documentation

APIs intended for widespread reuse are often reasonably well-documented [29]. The quality, quantity, and structure of this documentation can vary widely [30]. However, as code meant to be called and reused by unrelated client applications, documentation is often key to successful API uptake [30]. High-quality API documentation usually includes detailed descriptions of each function, method, and class, along with their expected inputs, outputs, and error conditions [31]. This information can be leveraged for classifying APIs, finding alternatives, or adapting usage when breaking changes are introduced.

In general, documentation may also contain examples and best practices [31, 32], which can be analyzed to infer latent properties of the APIs. These properties can then inform automated refactoring tools, ensuring that the transformation results in a use case aligned with the API's purpose. Furthermore, modern API documentation is increasingly enriched with metadata, such as annotations in dynamically typed programming languages [33]. This metadata can also be leveraged to guide the automated refactoring process.

In the case of breaking API changes, documentation evolves alongside the API itself [34], reflecting changes and deprecations in API behavior. If the documentation of evolving APIs provides transition examples from

old usage to new usages, or even natural language descriptions of how to adapt to the changes, then these can also be leveraged for automated refactoring.

In short, our hypothesis is that we can: *1. leverage API documentation to establish mappings between closely related APIs and libraries to facilitate API refactoring, 2. use examples and structured information from the documentation to guide the refactoring process, and 3. leverage metadata to increase the accuracy of the refactoring.*

### 1.2.2 API Development Process

Pull requests have become the *de facto* standard for software development on collaborative platforms like GitHub [35, 36]. In this method, a developer first clones the project (i.e., makes a personal copy of the project), makes changes in their copy, and finally submits these changes to the central repository for review. Pull requests typically include a title, a natural language description of the proposed changes, 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 decide whether to accept, request revisions, or reject the changes.

Pull requests involving API changes are also a rich source of data for mining transformation rules for API refactoring. Our hypothesis is that by examining pull requests (PRs) submitted to libraries where APIs are broken, we can identify and extract rules governing these changes. First, we can use tags in PRs to identify API changes by searching for labels such as "deprecated", "breaking change", and "API change". If the PR corresponds to such an API change, we can use self-contained commits in the PRs to learn code transformation rules for updating client code. The internal updates to the library source code resulting from the API change (such as test case changes) serve as the ground truth for mining rules and adapting client code.

### 1.2.3 Natural Language

Software development involves much more than writing code—it is surrounded by a rich ecosystem of natural language data, including commit messages, issue reports, discussions, and code comments [37]. Although unstructured, this information plays a critical role in guiding and informing automated API refactoring tools.

Developers frequently describe intended API changes in documentation and issue trackers [38]. Because of this, general-purpose large language models (LLMs) [39] trained on this data across many libraries are well-positioned to assist in addressing breaking changes and migration tasks.

Moreover, LLMs, through pretraining on large-scale code corpora, implicitly learn joint representations of APIs and the semantic relationships between them, including those across different libraries [40]. While this embedded knowledge can be noisy or incomplete, it is often sufficient to generate synthetic data for mining API migration patterns. Specifically, models can produce code snippets in both the source and target libraries, along with tests to validate their behavioral equivalence. Through this generate-and-test strategy, we can extract useful migration examples that form the backbone of automated migration tooling.

In addition to synthetic data generation, other natural language signals are also valuable during refactoring. Compiler error messages, for example, often highlight incorrect API usage and suggest potential fixes. Interpreting these messages can help automated tools converge more quickly on the correct transformations.

### 1.2.4 Program Synthesis

So far, our observations have highlighted alternative data sources that can be leveraged for automated API migration. In this section, we discuss how we will leverage them towards automated API refactoring. Specifically, we frame automated API refactoring as a program synthesis problem.

Program synthesis is a research area focused on automatically generating programs that comply with a given specification, such as natural language or input-output examples [41]. In our work, we frame API refactoring as a synthesis problem in two distinct ways:

1. **Direct Migration using Synthesis:** We approach the API refactoring problem by synthesizing programs directly. Specifically, given a program that uses library $A$, our goal is to generate an equivalent program that replaces all usages of $A$ with an alternative library $B$. Unlike the general setup in synthesis (such as programming-by-example [41]), our specification is complete, that is, the source program fully describes the intended behavior of the program to be synthesized.
2. **Script-based Migration Synthesis:** Instead of generating code per se, we also learn migrations more broadly and generalize them into edit or migration scripts that can be applied across multiple projects. For example, we can generate a script that automatically migrates projects from $A$ to $B$. In this work, we represent our scripts using declarative languages for matching and replacing code based on match-replace rules [5, 23].

## 1.3 Evaluation Methodology

We test our hypotheses by developing automated API refactoring methods based upon them. The evaluation of each method is as follows.

### 1.3.1 Metrics

To measure the effectiveness and efficiency of our techniques, we use the following metrics:

1. **Refactoring Accuracy**: This metric evaluates the quality of the refactorings produced by our tools. Specifically, for fixing breaking changes, we upgrade client projects to a new library version (e.g., $v_1$ to $v_2$) and run their tests post-upgrade both before and after refactoring. In the context of library migrations, we migrate the code and run the same test suite where feasible. If this is not possible, we use automatically generated tests.

2. **Runtime Performance**: We assess the runtime taken by our tools during refactoring. We also measure the impact of each component of our proposed approaches in ablation studies. Finally, we compare our tools to state-of-the-art alternatives where such comparison is possible and fair.
3. **Refactoring Rule Accuracy**: Validating refactoring rules is hard, as ground truth for the rule typically does not exist, and there are infinitely many programs that could spur different behaviors from the rule. Thus, instead, we manually validate the quality of the match-replace rules generated by our proposed approach. Our manual analyses always take into consideration the context and the intent of the rule. For rules to fix breaking changes, we examine documentation, developer comments, and online discussions to determine if our rule for addressing the breaking change preserves the original behavior. We use inter-rater agreement [42] as outlined by best practices for all our manual validations.

### 1.3.2 Benchmark Datasets

The benchmarks used vary depending on the underlying task. To assess refactoring accuracy and runtime performance, we rely on two types of benchmarks: real-world projects from GitHub and synthetic examples written in `python`. Each benchmark instance is accompanied by a set of test cases, some of which are automatically generated. These tests play a critical role in ensuring that the program's behavior remains consistent before and after refactoring. The benchmarks feature outdated API usages that require migration.

We choose to target `python` because it is the most widely used programming language today and represents a well-known gap in existing refactoring tooling [43].

To assess rule accuracy, we select relevant API migrations and breaking changes from open-source libraries, focusing particularly on ones where the author has direct expertise. We infer behavior-preserving refactoring rules and validate their correctness by manually checking whether the match-replace rules are generally applicable. This validation considers available documentation, developer discussions, and other public resources.

## 1.4 Contributions and Outline

We propose several techniques and prototype tools based on our ideas:

- **Synthesis-based Refactoring Using Documentation and Error Messages (Chapter 3):** A novel technique, named `SOAR`, that uses readily available API documentation to learn API representations and migrate code between libraries. SOAR uses program synthesis to automatically compute the correct configuration of arguments and necessary glue code for API invocation. It also integrates the interpreter's error messages to refine the search space during refactoring. This work was published in the proceedings of the *International Conference on Software Engineering* 2021 [44].

- **Mining API Migration Rules from Pull Requests (Chapter 4):** A novel technique, named MELT, that generates lightweight API refactoring rules for fixing breaking changes by using data from the pull requests that broke the API. The data is used in two ways: first, natural language descriptions and code changes in pull requests are used to generate adaptation examples. The examples are then tested and generalized into API transformation rules. Secondly, code changes to test cases within the library that were adapted to cope with the breaking change are also used to mine rules. This work was published in the proceedings of the *International Conference on Automated Software Engineering* 2023 [45].
- **A Multi-Language Code Transformation Tool (Chapter 5):** We develop a new declarative domain-specific language (DSL), called PolyglotPiranha, for expressing interdependent multi-language code transformations . The language aims to make lightweight transformation tools more expressive for complex refactorings. The language takes inspiration from other lightweight match-replace languages and enhances their design. We demonstrate the language and toolset effectiveness and expressiveness in an industrial setting, as well as its ease of use. This work was published at the *International Conference on Programming Language Design and Implementation* 2024 [5].
- **Distilling code migration knowledge from LLMs (Chapter Chapter 6):** We introduce a hybrid approach that leverages large language models to extract concrete API migration examples and then generalizes these into reusable, rule-based transformation scripts in the PolyglotPiranha language. This method transforms the LLM's unstructured migration knowledge into testable and repeatable migration logic, eliminating the need for existing migration corpora or manual effort. This work is under submission.

*This work is a compilation of the papers discussed above.*

# 2

# Research Background and Related Work

## Contents



In this chapter, we delve into two fundamental concepts to this work and to modern software development in general: Application Programming Interfaces (APIs) and refactoring. We start by offering a concise overview of APIs, discussing their role in creating re-usable software systems (Section 2.1). Following this, we overview the practice of refactoring in software engineering, a key process for enhancing code quality (Section 2.2). We then address the challenges developers encounter during refactoring, emphasizing the need for automated solutions (Section 2.3). We also give an overview of how researchers have developed automated approaches for API refactoring, highlighting trends and gaps in this area (Section 2.4). Finally, we review recent work leveraging large language models (LLMs) for source code transformation tasks, particularly refactoring and library migration (Section 2.5).

## 2.1 Application Programming Interfaces

Developers oftentimes build software meant for use by other software, rather than directly by end-users. An important consideration in such cases is to decide which functionality is intended to be provided and the abstractions to be exposed to developer clients. According to design best practices [46], there must be a clear separation between the software's functional requirements (what it does) and its concrete implementation (how it does it). This separation is critical to ensure that clients are not burdened with irrelevant implementation details. The functionalities made available to clients are collectively known as an Application Programming Interface, or API.

The term API has been loosely used to describe a wide array of concepts, interpreted differently across technical domains [47]. Indeed, there is considerable debate about its meaning and usage [48]. However, in this work, we use the term 'API' to denote a software API, typically exposed and implemented as a library with a collection of classes, functions, or methods (depending on the programming language) that expose certain functionalities for other programmers to use. A single API can have multiple implementations (or none, if abstract) in the form of bindings that share the same programming interface. For example, because Scala and Java compile to compatible bytecode, Scala programs may use APIs implemented in Java.

Since APIs are intended for reuse, they often come with documentation, describing classes, methods, and sometimes typical usage cases, design rationales, and performance discussions [49, 50]. Regardless of what constitutes a particular API, the important underlying concept is that an API is a well-defined interface providing specific services to other software components.

APIs are the cornerstone of modern software engineering. Modern applications are often built on top of many APIs, which are also built upon other APIs. This approach yields significant productivity gains, allowing developers to focus on their specific tasks. Various research studies have investigated API design [51], evolution [52], and usability [53].

## 2.2 Refactoring

Real-world software is constantly evolving, often driven by the need for enhancements and changes to meet new requirements [54]. As software adapts to additional functionality, code complexity tends to increase, and the software drifts from its original design. This issue is further exacerbated when developers prioritize short-term fixes over comprehensive, long-term solutions. While these immediate solutions might initially appear cost-effective, they frequently result in significant, hidden long-term maintenance costs. Such escalating complexity eventually leads to a gradual deterioration, a phenomenon known as *technical debt* [11]. The longer technical debt remains unaddressed, the more 'interest' it accrues, making it increasingly challenging to develop new features and maintain existing code.

Providing accurate estimates for software maintenance spending is challenging, but it is widely accepted

that the expenses related to software maintenance — including bug fixes, design enhancements, and code restructuring — vastly exceed the cost of actual new feature development. Complex code often results in extended development periods, subtle bugs, and increased cost of change [55]. Additionally, according to Minelli et al. [56], developers allocate approximately 70% of their time to understanding code rather than coding. Therefore, a well-thought-out design and adherence to best practices are crucial to reduce this effort [57].

One way to tackle this spiral of complexity is through a software engineering practice known as *refactoring*. Refactoring is defined as *the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure* [9]. Sometimes, refactoring is colloquially used in a more generic and less rigorous manner than this definition suggests, and indeed, some authors argue that refactoring is not always behavior-preserving [58]. However, in this document, we are primarily concerned with behavior-preserving refactorings, aiming to improve maintainability without external changes, as refactorings that do not preserve behavior have limited potential for full automation (i.e., cannot be tested).

In general, a refactoring is parameterized as a sequence of program transformations applied either manually or automatically with a tool, along with a specification that guarantees behavior preservation if satisfied. In modern software development, tests are the most widely adopted form of specification. Formally, we run a given program on a set of predefined inputs and then check if the observed output matches the expected output. Although refactoring can be applied in different paradigms, we focus on imperative programming languages. Many forms of refactoring also exist, like renaming variables, moving methods, migrating across languages [59], or refactoring APIs [60] (the primary focus of this work).

There are two primary API refactoring tasks of interest: library migration and library updates. A *library migration* involves replacing a third-party library (the source library) in a project with an alternative one (the target library). In the case of a *library update*, the task is to replace an old version with its newer iteration. Numerous factors can drive such migrations or updates, including (but not limited to) the introduction of new features, performance enhancements, improved community support and documentation, compatibility issues, license changes or restrictions, and deprecation, among others.

## 2.3 Challenges in Refactoring

The implementation and frequency of refactoring practices vary and are influenced by several factors.

**Perception and Management Challenges.** The majority of developers recognize the need for continuous refactoring in the development life cycle [61]. However, developers also report that a significant barrier to refactoring is lack of time, as they often need to prioritize immediate functional tasks over enhancements in source code quality, either due to pressing deadlines, high workloads. [61]. A major obstacle to refactoring implementation is the widespread perception that it slows down the development of new features [62]. This

view poses a challenge in securing management support for refactoring initiatives, as the tangible benefits of refactoring are often long-term and not immediate [63].

Other studies corroborate this issue, showing that refactoring, especially API refactoring, tends to be delayed and postponed, leading to future compatibility challenges and maintenance overheads [26]. This creates a paradox: while refactoring is intended to simplify development processes and reduce the cost of changes, the prevailing industry practices and perceptions often prevent its effective implementation.

Refactoring in a small, single-developer codebase is hard enough; performing refactoring in large, industrial codebases with many developers introduces additional socio-technical complexities. Large projects entail coordination across teams, agreement on design changes, and careful sequencing of refactorings to avoid disrupting ongoing work. Kim et al. [14] reported that "the need for coordination with other developers and teams" was frequently mentioned as an inherent challenge when refactoring at Microsoft. Because one team's refactoring may impact modules owned by other teams, extensive communication and planning are required. In agile environments, this is often at odds with fast iteration cycles, and in more plan-driven environments, it may require bureaucratic change control. Either way, the overhead of coordination can discourage refactoring altogether unless the benefits are very clear. Moreover, in distributed teams or large organizations, not everyone may agree that a given refactoring is worthwhile, leading to social barriers. For instance, a developer might refrain from improving a messy component if its original author or owner is resistant to changes.

Interviews from the Windows refactoring effort revealed that having a designated refactoring team helped bypass some of these issues, as that team had a mandate to make systemic improvements and could negotiate changes across component boundaries [14, 58]. However, most projects do not have such dedicated refactoring task forces; instead, individual developers must champion and justify each non-trivial refactoring, which is a socio-technical negotiation that many prefer to avoid.

**Risks Associated with Refactoring.** In addition to time constraints and management challenges, developers also perceive refactoring as a task fraught with risks. A recurring concern is the fear of inadvertently introducing bugs or causing regressions in existing functionalities. In a field study with Microsoft engineers, Kim et al. [14] found that 76% of the participants acknowledge that refactoring is a task that leads to subtle bugs and regressions. This perception has been observed and confirmed in other studies [64], noting a correlation between API refactoring and an immediate increase in bugs. Moreover, developers also reported concerns related to extra time required for code reviews, and dealing with merge conflicts, as well as risks of over-engineering during the refactoring process. These risks add to the perception of refactoring as a fault-prone activity, as highlighted in other studies [65].

Refactoring can also incur the cost of re-testing and re-validation. Even if no bugs are introduced, significant refactorings demand re-running the test suite and performing thorough reviews to ensure behavior is unchanged. In the Microsoft field study [14], about 24% of developers pointed out that refactoring increases

testing overhead, as all regression tests must pass after the changes. Many engineers reported that without adequate tests, they simply "have no safety net" and thus avoid refactoring. One interviewee advised that "if there are no tests… refactoring should not be done" given the inability to catch errors.

**Lack of Tool Support and Automation.** Empirical studies agree that the refactoring commands shipped with modern IDEs are far from routine practice. Kim et al. [14] report that developers perform about 86 percent of their refactorings by hand, and fully half of the respondents never invoke an automated command, even when available. Participants cite three main reasons: tool edits are hard to merge and review, the engines operate at too low a level of abstraction, and there is little support for confirming behavior preservation.

A survey by Vassallo et al. [61] also ranks the absence of automatic refactoring tooling as the third most common barrier to continuous refactoring, mentioned by 16.7% of participants [61]. A recent systematic review of API evolution literature lists improving the performance and usability of refactoring and migration tools among the key open research challenges that must be addressed before widespread adoption is possible [27]. Industrial reflections echo the same need, emphasizing that robust automation is a prerequisite for treating refactoring as a regular engineering task [66].

Another aspect of tool inadequacy is poor integration with other development workflows. Tools often fail to support the broader refactoring process, such as incorporating large refactoring changes into version control, code review, and continuous integration. Developers voiced the need for refactoring-aware code review and merge tools [14]. Tempero et al. [67] report that source control systems were cited as a barrier: one respondent noted that "source control makes [a large refactoring] quite painful," as rename and move operations do not map cleanly to line-based diffs, obscuring change history. This pain point can make teams reluctant to accept refactoring contributions, as they complicate blame analysis and branch merges. Moreover, automated tools sometimes fail to guarantee behavior preservation in practice, undermining developer trust. Murphy-Hill and Black [68] observed that many refactoring tools were not fully "fit for purpose," i.e., they did not meet the principles of "safe, frequent, and unobtrusive" refactoring. For example, some tools do not allow easy intermixing of small refactorings with other edits, which is how developers often work. Such usability limitations may help explain the empirical finding that developers abort or undo a significant fraction of automated refactoring operations [69].

Together, these findings point to a clear research gap: we still need refactoring engines that operate on higher-level program concepts, integrate smoothly with version control, testing, and review workflows, and provide developers with immediate evidence that intended behavior is preserved, along with realistic effort–benefit estimates.

## 2.4 Automated Techniques for API Refactoring

The challenges associated with refactoring have motivated multiple research efforts in automation. In this section, we provide a brief overview of key ideas and trends in automated API refactoring.

### 2.4.1 Code Transformation Languages

As discussed in Section 2.2, a refactoring is parameterized as a sequence of program transformations. For large-scale, automated refactoring, it is necessary to represent this sequence of transformations in a language. Here, we give a brief overview of code transformation languages proposed over the years.

Code transformation languages and toolsets can be divided into declarative and imperative.

#### 2.4.1.A Declarative Code Transformation Toolsets

Declarative languages for code transformations work with find-replace rules; each rule has two parts: (1) a template for selecting the source code to be transformed, and (2) a replacement template that shows how the matched code should be transformed. Declarative match-replace toolsets can either be language-specific or language-agnostic. Language-specific toolsets include `Coccinelle` [70], widely adopted in the Linux community for C, and `Refaster` [71]. Variants of these tools include `Coccinelle4J` [72] for Java and `GoPatch` [73] for Go. Another well-known toolset is `libclang` [74] and its AST matchers. `libclang` provides a declarative language for searching over code using AST matchers and other predicates, which can then be rewritten imperatively using the library's rewrite API. All these tools are language-specific, allowing them to leverage existing compiler infrastructure and semantic information, such as control flow, to enable precise transformations. However, a significant disadvantage is that substantial effort is required to introduce and maintain new language front-ends for these tools because of this reliance on compiler infrastructure.

In contrast, lightweight tools like `comby` [23] and `ast-grep` [75] present alternatives that rely on ad-hoc simple grammars and parse trees without name resolution or deeper language understanding. This allows them to support multiple languages with minimal maintenance overhead. For example, `comby` only requires users to define a simple Dyck-extended grammar, specifying which constructs are used for brackets, comments, etc. Nonetheless, lightweight declarative tools have their own limitations. Since they generally have limited understanding of code context and semantics, it becomes challenging to express non-atomic transformations that rely on inspection and semantic information.

Language workbenches, like Spoofax [76] (which incorporates Stratego [77]) and Rascal [78], offer comprehensive toolsets for designing and implementing languages, including metalanguages for writing code transformations. Similarly, DMS [79] provides a declarative rewrite language for transforming code and allows users to combine it with procedural rewrites. However, these tools require the specification of the target language's grammar and its extension to support metasyntax using the toolset.

TXL [80] is a multi-language transformation tool that requires users to write both grammar specifications and transformation rules within the TXL language. TXL transformations often use non-terminal symbols in rewrite rules, which makes them non-trivial to write and use. Cubix [81] provides an alternative multi-language solution using compositional data types. Cubix, as described by its authors, is not intended for the lay programmer and requires significant effort and expertise to learn.

Query languages for code that support complex analyses, namely CodeQL [82], can also be used for searching code. This enhances the precision of rule matching with semantic awareness and can be useful in some scenarios. Note, however, that CodeQL has limited language support, may introduce performance overheads, and can require additional integration effort due to its build system dependencies.

#### 2.4.1.B Imperative Code Transformations

Researchers and tool builders have also invested heavily in the development of advanced imperative refactoring engines [83–86], migration [87–89], and cleanup tools [90], such as ErrorProne [91] or OpenRewrite [92]. In particular, OpenRewrite (which is language-specific) offers a framework based on the visitor pattern to implement transformation recipes. Powerful imperative frameworks are also usually built around compiler infrastructure and can leverage symbolic information (e.g., name resolution) and other semantic context. Their usage is justified in cases where in-depth analysis is needed. However, not all real-world code transformations require such heavyweight infrastructure (as we will see in Chapters 4 to 6).

It is also possible to do source code manipulation with parser generator libraries (e.g., `tree-sitter` [93]), which avoids the problem of deep integration with compiler infrastructure. `tree-sitter` provides many community-maintained grammars and multiple front-ends for over 100 languages, as well as a common interface between all of them, making it ideal for general source code manipulations. Indeed, `tree-sitter` is used as the backend for multiple refactoring engines (including our declarative language `PolyglotPiranha`) presented in Chapter 5, and other alternatives like `ast-grep` [75]. `tree-sitter` serves as a foundation for building language-agnostic but syntax-aware refactoring engines with minimal overhead.

### 2.4.2 Learning API Transformations

Code transformation languages and toolsets allow us to manipulate source code, but they do not automate the refactoring process. In this section, we review how researchers have partially automated API refactoring tasks over the years.

#### 2.4.2.A Mining to Learn Edit Scripts

Automating API refactorings requires automatically inferring transformation data from reliable sources. Prior research has primarily focused on mining API clients to learn these rules. The key idea behind these approaches is to locate API clients on collaborative online coding platforms such as `GitHub` [24] and analyze

their version history (i.e., commits). The goal is then to identify commits where clients have either migrated to a newer API version or updated their API usage. This data is aggregated and used to mine refactoring rules using various algorithms.

The majority of approaches to mining rules have primarily targeted APIs in object-oriented languages (specifically, Java and C#). Examples include `A3` [19] and `Meditor` [21]. Although these tools use various techniques for the mining process and adopt different internal representations to express the refactorings, their approaches are based on similar ideas. Some tools represent refactorings as sequences of edits on structure rather than as match-replace rules. For example, `APIFix` [18] mines transition examples from both previously-migrated and new client repositories to learn refactoring actions. `APIFix` represents code transformation as a sequence of tree edits using `Refazer`'s [94] program synthesis engine, rather than using regular match-replace rules. Abstract Syntax Tree (AST) edits are typically more challenging to understand [95].

`APIMigrator` [22] and `AppEvolve` [20] also mine client repositories for commit data to generate refactoring rules and apply them to clients. A key difference in these tools compared to prior work is that they also leverage differential testing techniques [96] to validate edits on clients, rather than only checking the syntactical validity of transformed code.

A major issue with existing mining approaches is that data for mining is often very scarce, which limits the applicability and usefulness of these approaches. A recent study by Kula et al. found out that 81.5% of projects keep outdated dependencies. Additionally, the mining process can occur only after the clients begin to transition between versions, preventing use shortly after a new version of the library is released [26, 27].

#### 2.4.2.B Mining API Mappings

Another line of research focuses on discovering mappings between old and new APIs (i.e., "what to replace" in a migration). Instead of learning how to transform the code, these approaches tackle the problem of establishing mappings between a source and a target APIs. These approaches mine API mappings, i.e., pairs of functions, classes, or other API elements that serve analogous roles across libraries or versions. For example, `SemDiff` [98] infers mappings between two versions of a library by examining client code: if an API element is removed or changed in a new version, `SemDiff` looks for usages of the old element in client projects and correlates them with newly introduced or alternative API calls after the library update. In this way, it recommends likely replacement methods or classes for a given deprecated API. Such mapping inference can identify one-to-one replacements (e.g., method `OldX` is replaced by `NewY`) with high recall. Other approaches use more sophisticated analyses to filter candidate mappings (for example, `PART` [99]), aiming to reduce false positives. However, mapping-only techniques generally do not describe how to adapt the rest of the code, focusing solely on the API call substitutions.

Researchers have also explored doing cross-language API mappings. For example, `StaMiner` [100] and

MAM [59] learn equivalences between libraries in different programming ecosystems by leveraging large corpora of "bilingual" projects or code that has been ported. MAM (Mining API Mappings) analyzes projects that have both Java and C# versions to statistically infer which Java API calls correspond to which C# API calls providing similar functionality. StaMiner extends this idea by incorporating usage context and frequency information to improve the confidence of each mapping; its output has been used to enhance an existing Java to C# transpiler (Java2CS [101]), to improve the quality of the transpiler's output.

Recently, semantic embedding techniques have been applied to this problem: for instance, DeepAM [28] embeds API elements and their usage contexts into a vector space to automatically find "analogous APIs" across different libraries or frameworks, much like we do in Chapter 3. Such an approach can uncover mappings even when simple naming or structural cues are absent, by relying on learned semantic similarities in how the APIs are used. In general, mining API mappings tackles the migration problem at the level of identifying the correct target API elements. These techniques excel at suggesting what new API corresponds to a given deprecated one, and they have shown high effectiveness in finding correct replacements for numerous legacy APIs. Nevertheless, as noted by prior studies, knowing the mapping alone is often insufficient for complex migrations that require additional code adaptations (beyond a direct call replacement) [19].

#### 2.4.2.C Example-based approaches

Instead of taking existing examples (e.g., from GitHub) to learn transformation rules, researchers have proposed *refactoring-by-example* techniques. These approaches infer the transformations as a program in a low-level DSL from input-output examples. The idea is for the developer to provide an example which is then generalized, rather than mining from client projects. For example, LASE [102] and Bluepencil [103] infer an edit script from two example edits. More recently, Overwatch [104] integrates refactoring by example ideas into core IDE infrastructure to learn edit sequences, not just from input-output examples but also from intermediate steps (i.e., using temporal context). Catchup! [105] follows a different approach. Instead of asking users for examples, it asks library developers to record refactoring operations. The idea is to provide a patch alongside the new library version so that clients can automatically update their dependencies.

By learning from concrete usage scenarios, example-based approaches can tackle more complex edits (e.g., reordering calls, inserting new method invocations, or other major refactoring) that pure mapping cannot represent or infer from version histories. The main idea of the example-based approach is to keep the developer in the loop and iterate on the change until it arrives at a correct solution.

## 2.5 Large Language Models

Large-scale language models (LLMs) pretrained on vast corpora of source code and natural-language text have rapidly become a foundational technology for automated software engineering tasks, including auto-

mated refactoring and library migration [39, 106–110]. This section reviews how LLMs are being applied to support code transformation, library migration, and other refactoring activities.

LLMs have demonstrated the ability to perform non-trivial code refactorings and transformations given natural language instructions. For example, they can identify and apply common refactoring operations in code, such as renaming methods or extracting methods, by following prompts describing the desired change [111, 112]. In an empirical study on automated refactoring, Liu et al. [113] evaluated LLMs on 180 real-world Java refactoring tasks. They found that with proper guidance (e.g., explaining the refactoring type and narrowing the search scope), models would complete simple refactoring actions with a success rate of 86.7% on their benchmark.

Despite these promising results, LLM-based refactoring is not yet fully reliable. In the same study, Liu et al. [113] report that a subset of LLM refactoring suggestions (approximately 7% of them) had subtly introduced bugs, either by altering program behavior or introducing syntax errors. Indeed, it is well known that using LLMs to transform code directly is generally opaque and lacks formal guarantees of correctness, which can result in subtle bugs [114]. To mitigate such risks, researchers have proposed hybrid approaches that combine LLM intelligence with traditional verified transformation tools. For instance, the `RefactoringMirror` [113] technique replays the refactoring edits proposed by an LLM through a deterministic engine (like an IDE's refactoring tool) to validate and apply only behavior-preserving changes, similar to `CatchUp!`, where human-made refactorings are recorded and then replayed [105].

Researchers have explored both training and zero-shot prompting paradigms for leveraging LLMs in code migration tasks. `DeepMig` [115] introduces a transformer-based model that is trained on paired examples of real-world API migrations to learn how to jointly recommend library upgrades and transform dependent code. However, this approach requires curated training data from projects that have undergone such migrations.

In contrast, Islam et al. [116] show that prompting an LLM can be surprisingly effective for library migration in Python, even without any additional fine-tuning. Their method relies on instructing the model to replace the usage of one library with another and explain the changes. Their study migrated 321 real-world instances of library replacement (covering 2.9K line changes) and found that the LLMs could correctly apply 89–94% of the code modifications needed compared to ground truth migrations. However, note that these results need to be interpreted cautiously due to data leakage [117] since the benchmark consists of open-source migration examples publicly available on GitHub. In terms of functional correctness, around 36–64% of the LLM-migrated projects passed all existing unit tests, depending on the model (with larger `GPT-4` variants performing best). These results suggest that LLMs can learn not only to map APIs but also to adjust surrounding glue code, such as import statements, data types, and error handling.

Recent reports have also shown the viability of LLMs for source code manipulation in industrial applications. Commercial tools such as `Amazon Q Developer` [118] leverage LLMs to automate framework and language version upgrades (e.g., Java 8 to Java 17), applying large-scale transformations across codebases.

Similarly, Google has reported success using LLMs to assist in large-scale migrations such as test framework updates and legacy API deprecation [119], integrating these models with tools like `OpenRewrite` for downstream integration and verification.

Overall, while LLMs can automate a wide range of source-to-source transformations, they need careful oversight or coupling with rule-based systems to avoid introducing bugs. In our work, we show ways to integrate LLM capabilities for reliable API refactoring.

# 3

# Synthesis-Driven Refactoring with Documentation and Error Messages

## Contents



In this chapter, I discuss my completed and published work on a "Synthesis Approach for Open Source API Refactoring" (SOAR) [44]. In this work, we leverage: 1. API documentation (including metadata), and 2. interpreter error messages to automatically refactor APIs. Unlike previous approaches, SOAR does not require any pairwise training data to find API mappings. Moreover, SOAR migrates code using a program synthesis based approach rather than using refactoring rules. This makes SOAR more powerful and expressive, as some transformations cannot be expressed as match-replace rules. SOAR is one the first end-to-end, synthesis-based approaches to API refactoring requiring minimal to no training data. In the evaluation, we show that it can

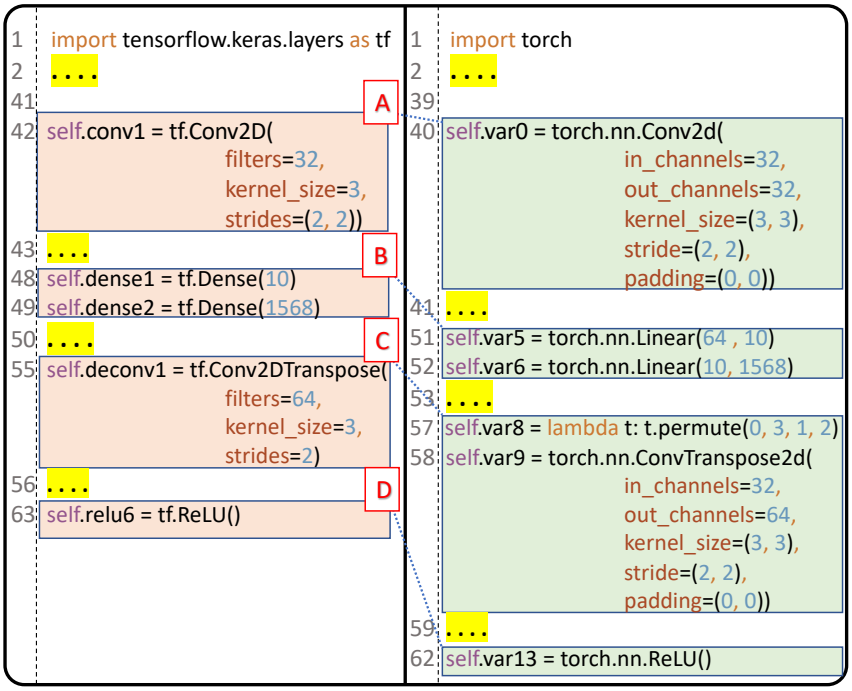



**Figure 3.1:** An example of how SOAR refactors a program written with `TensorFlow` (left) to using `PyTorch` (right). Note that the whole program consists of 15 APIs calls to `TensorFlow`, though we only show four blocks of them (i.e., A, B, C and D) for brevity. SOAR can migrate the full program in 161 seconds.

successfully migrate functions / programs of up to 45 lines within reasonable time frames.

## 3.1 Motivating Example

We illustrate some of the difficulties of manual API refactoring via example. Consider the code snippet depicted on the left-hand side of Figure 3.1. This code snippets features an autoencoder, a specific type of neural network, developed using the `TensorFlow` API. Our objective is to transition this code to the `PyTorch` API as shown in the right-hand side. For context, an autoencoder is an encoder-decoder style neural network designed for data compression. It is trained to transform data into a more compact form (i.e.,, a latent representation), and then reconstruct the original data as accurately as possible.

The example in Figure 3.1 shows only a portion of the program, for didactic purposes. To build the first layer of the encoder, it calls the `Conv2D` function, creating a convolution layer for 2D images. After further (elided) activation and convolution layers, it calls `Dense` to output a latent representation of the input image. Decoding this output follows roughly the same procedure as the encoding, but using `Conv2DTranspose` instead of `Conv2D`. The function `ReLu` appears in both the encoder (not shown) and decoder, is used to ensure non-linearity of the neural network.

The example in Figure 3.1 illustrates several of the core challenges in refactoring open-source APIs, as well

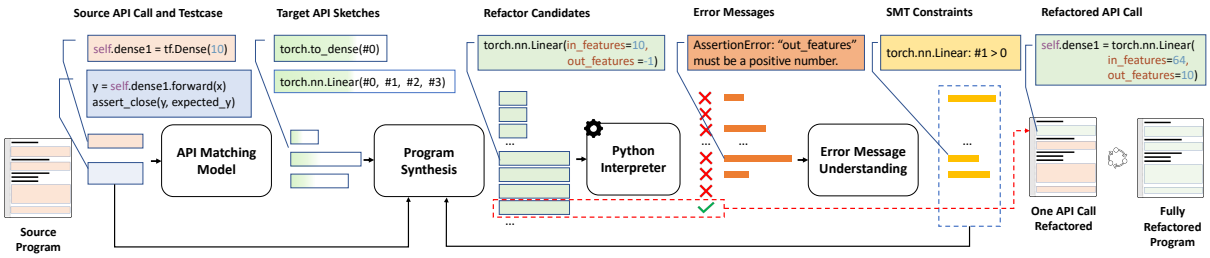



**Figure 3.2:** Overview of SOAR's architecture.

as opportunities to inform an automated approach. First, the names of function calls implementing similar functionality may be very similar or even identical (such as those in blocks A, C, and D), or completely different (e.g., `Dense` versus `Linear` in block B). If a developer were performing this migration manually, they might reference the API documentation. For example, `TensorFlow` documentation describes the `Conv2D` class as a 2D convolution layer (e.g., spatial convolution over images)" [16]; the corresponding PyTorch documentation for the `Conv2d` call describes it similarly, as a 2D convolution over an input signal composed of several input planes" [120]. Here, the function names map well, but when this does not happen, the documentation should at least provide analogous descriptions for functions offering equivalent functionality.

Identifying the appropriate function is only part of the challenge in API refactoring. Even when the correct function is known, APIs mapped for the same functionality may have parameters with different names, types, conventions, and default values. This is evident in the majority of calls in our example (as seen in blocks A, B, and C). For instance, the `Conv2D` functions in both libraries take different parameters. There is some overlap — both include `kernel_size`, and `stride` corresponds with `strides` — yet they may expect different types (for example, `kernel_size` takes an integer in TensorFlow but a tuple in PyTorch). In some cases, new arguments must be inferred, varying based on context. For example, the first parameter of the `Linear` API calls in block B, mapped from `Dense`, require an extra argument to be dynamically computed. This makes it infeasible to write a simple match-replace rule for `Dense` to `Linear` mapping, as arguments must be dynamically generated. Finally, there are instances where no single function in the target API matches the semantics of a call from the source API, necessitating a one-to-many mapping, as illustrated in the conversion of the `Conv2DTranspose` call in block C.

## 3.2 SOAR's Algorithm

This section describes `SOAR`, our approach for automatic API migration. We begin with an overview of the method (Section 3.2.1) before providing more detail on individual components (Section 3.2.2; 3.2.3; 3.2.5).

**Algorithm 1** SOAR($\mathcal{I}, \mathcal{S}, \mathcal{T}, \mathcal{C}$)

**Input:** $\mathcal{I}$: existing program, $\mathcal{S}$: source library, $\mathcal{T}$: target library, $\mathcal{C}$: test cases
**Output:** $\mathcal{O}$: refactored program
1: $\vec{r}$ : API mapping = MAPAPI($\mathcal{T}, \mathcal{S}$)
2: $\mathcal{O} = \{\}$
3: **for each** $l \in \mathcal{I}$ **do**
4: $\quad \mathcal{O} = \mathcal{O} \cup$ REFACTORLINE($l, \mathcal{T}, \mathcal{C}, \vec{r}$)
5: **end for**

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (*string, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

**Figure 3.3:** Description of the program parameters in `torch.nn.Conv2d` documentation [120].

### 3.2.1 Overview

Figure 3.2 shows an overview of the SOAR architecture, while Algorithm 1 provides an algorithmic view. SOAR takes as input a program $\mathcal{I}$ consisting of a sequence of API calls from a source library $\mathcal{S}$, the source ($\mathcal{S}$) and target ($\mathcal{T}$) libraries and their corresponding documention, and a set of existing test cases ($\mathcal{C}$). Since the user wants to refactor code from $\mathcal{S}$ to $\mathcal{T}$, we assume that the user already has test cases for $\mathcal{I}$ that can be reused to check if the refactored code ($\mathcal{O}$) has the same functional behavior has the original code ($\mathcal{I}$). Refactoring proceeds one line at a time in $\mathcal{I}$, finding/constructing an equivalent snippet of code (composed by one or more lines) that uses APIs of the target library $\mathcal{T}$; the composition of all these translated lines comprises the output $\mathcal{O}$.

For each API call in the input program, the first problem either a developer or a tool must face is to identify methods in the target API that implement the same functionality (i.e., for a given set of input parameters, the target API call must generate the same output). SOAR uses an *API matching model* to identify target API calls. This model is built using NLP techniques that analyze the provided API documentation for each call, and provides a mapping ($\vec{r}$ in Algorithm 1) that computes the similarity between each target API function and each potential source API function. SOAR uses this to find the most likely replacement methods in the target API for each source API call in the input program. We provide additional detail in Section 3.2.2.

Given a potential match call in the target API, the next step is to determine how to call it, in terms of providing the correct parameters, in the correct order, of the correct type. SOAR uses program synthesis to automatically write the refactored API call, using the provided test cases to define the expected behavior of the synthesized code and its constituent parts. The synthesis process can be assisted with additional automated analysis of API documentation, which often provides key information about each parameter, namely (1) whether it is required or optional, (2) its type, (3) its default value (if applicable), and (4) constraints between arguments, input and output (e.g., input and output tensor shapes). Figure 3.3 shows a snippet of the descriptions of all parameters for `torch.nn.Conv2d`. For example, the parameter `stride` is optional; it takes type `int` or `tuple`, and its default value is 1. Analysis of this documentation can produce a specification constraint for the `stride` parameter, assisting the program synthesis task. Section 3.2.3 describes the synthesis step.

Given a potential rewrite in the target API, a natural step for a developer would be to run the refactored code on test inputs. Unsuccessful runs can be quite informative, because many APIs (especially in the deep learning and data science domains) provide error messages that can be very helpful for debugging. SOAR simulates the manual debugging process by first adapting the input whole-program test cases to test partially refactored code, and then extracting both syntactic and semantic information from any error messages observed when running them. SOAR uses this information to add new constraints to the iterative synthesis process (Section 3.2.5).

After migrating all calls in the source API to the target API such that all input tests pass, SOAR outputs a fully refactored program. Subsequent sections provide additional detail on the previously described steps.

### 3.2.2 API Matching using Documentation

The first step in migrating a call in a source API is to identify candidate replacement calls in the target API with similar semantics. SOAR's API matching model ahcieves by analyzing the prose documentation associated with the APIs, rather than mining client projects.

At a high level, the model embeds each API method call in a source and target library into the same continuous high-dimensional space, and then computes similarity between two calls in terms of the distance between them in that space. We explored two information retrieval approach to obtain latent API representation: TF-IDF (term frequency – inverse document frequency) [121] and pretrained word embeddings [122].

**TF-IDF.** TF-IDF finds the most representative words in a sentence (which are usually different from the most frequent ones). The core idea is to discard irrelevant words when computing the API representation. For example, words like "the" or "this" convey very little information about an API.

For our TF-IDF model, we first derive a bag-of-words representation $\mathbf{x^i}$ from a description of an API call after some stemming of the words with the Snowball Stemmer [123]. $\mathbf{x^i} = [x_1^i, x_2^i, ..., x_n^i]$ where $x_j^i$ denotes the frequency with which word $x_j$ appeared in the sentence $\mathbf{x^i}$, and $n$ is the size of the vocabulary from the descriptions of all APIs considered. A TF-IDF representation of the call is computed as Equation (3.1):

$$\text{TF-IDF}(\mathbf{x^i}) = \left[ \frac{x_1^i}{\sum_{t=0}^{m} x_1^t}, \frac{x_2^i y}{\sum_{t=0}^{m} x_2^t}, \cdot, \frac{x_n^i}{\sum_{t=0}^{m} x_n^t} \right] \tag{3.1}$$

However, the major downside of TF-IDF is that it does not encode the similarities between words themselves. For example, consider two hypothetical call descriptions: (1) *Remove the last item of the collection*, and (2) *Delete one element from the end of the list*. They are semantically similar but since they have minimal overlapping words, a TF-IDF representation method would not recognize these two API calls as similar.

**Tfidf-GloVe.** We can fix this problem by adding the use of pretrained word embeddings. Specifically, we use the GloVe embedding [122], which is trained on a very large natural language corpus and learns to embed similar words closer in the embedding space. Since the paper was published many other embeddings models have emerged but they are fundamentally built using the same ideas.

To obtain sentence embeddings from individual words, we perform a weighted average of the word embeddings and use the TF-IDF scores of individual words as weight factors. It is a simple yet effective method to obtain sentence embedding for downstream tasks, as noted by previous work [124, 125]. This is shown in detail as Equation 3.2, where $\mathbf{w_j}$ is the vector encoding the GloVe embedding of word $x_j$:

$$\text{Embedding}(\mathbf{x^i}) = \sum_{i=j}^{n} \frac{x_j^i \cdot \mathbf{w_j}}{\sum_{t=0}^{m} x_j^t} \tag{3.2}$$

By including the GloVe embedding, word similarity is preserved; by including the TF-IDF terms, the influence of embeddings of common words is greatly reduced. However, GloVe is trained with Common Crawl [126] which contains raw webpages, which is a mismatch from our domain of textual data (i.e., data science and programming). This causes a lot of OOV (out-of-vocabulary) problems.

**API matching.** Given the representation of two APIs $\text{Rep}(\mathbf{x}^i)$, $\text{Rep}(\mathbf{x}^j)$ in the same space $\text{Rep}(\cdot)$, we compute their similarity with cosine distance:

$$\text{sim}(\text{Rep}(\mathbf{x}^i), \text{Rep}(\mathbf{x}^j)) = \frac{\text{Rep}(\mathbf{x}^i) \cdot \text{Rep}(\mathbf{x}^j)}{|\text{Rep}(\mathbf{x}^i)||\text{Rep}(\mathbf{x}^j)|} \tag{3.3}$$

For computational efficiency, we pre-compute the similarity matrix between the APIs across the source and target library. So we will be able to query the most similar API for the synthesizer to synthesize its parameters on the fly.

### 3.2.3 Program Synthesis

Instead of crafting match-replace rules using the API mappings, we refactor client code directly using program synthesis. This allows our refactoring engine to support expressive refactorings. Formally, given input test cases and an API matching model providing a ranked list $\vec{r}$ of APIs in the target library, the synthesis model automatically constructs new, equivalent code, of one or more lines, that uses APIs of the target library $\mathcal{T}$.

**Algorithm 2** REFACTORLINE($l, \mathcal{T}, C, \vec{r}$)

**Input:** $l$: line of code from $\mathcal{I}$, $\mathcal{T}$: target library, $C$: test cases, $\vec{r}$: ranked list of API matchings
**Output:** $\mathcal{R}$: refactored snippet

```
1:  O = {}
2:  for each l' ∈ r⃗ do
3:      s⃗ = GENERATESKETCHES(l', T)
4:      for each s ∈ s⃗ do
5:          R = FILLSKETCH(s)
6:          if PASSTESTS(R, C) then
7:              return R
8:          end if
9:      end for
10: end for
```

The refactored program $\mathcal{O}$ has the same functionality as input program $\mathcal{I}$, and passes the same set of tests $C$.

To refactor each line of the existing program $\mathcal{I}$, we use techniques of programming-by-example (PBE) synthesis [41]. PBE is a common approach for program synthesis, where the synthesizer takes as specification a set of input-output examples and automatically finds a program that satisfies those examples. In the context of program refactoring, our examples correspond to the test cases for the existing code. For our experiments we restrict ourselves to straight-line code where each line returns an object that can be tested. With these assumptions, we can automatically generate new test cases for each line $k$ of program $\mathcal{I}$. This can be done by using the input of the existing tests, running them, and using the output of line $k$ as a new test case for the program composed by lines 1 to $k$.

Our program synthesizer for refactoring of APIs is presented in Algorithm 2 and it is based on two main ideas: (i) program sketching, and (ii) program enumeration. For each line $l$ in program $\mathcal{I}$, we start by enumerating a program sketch (i.e., program with holes) using APIs from the target library $\mathcal{T}$ (line 3). For each program sketch, we perform program enumeration on the possible completion of the API parameters (line 5). For each complete program, we run the test cases for the program up to line $l$. If all test cases succeed, then we found a correct mapping for line $l$ between libraries $\mathcal{S}$ and $\mathcal{T}$ (line 6). Otherwise, we continue until we find a complete program that passes all test cases.

**Program Sketching.** Program sketching is a well-known technique for program synthesis [127] where the programmer provides a sketch of a program and the program synthesizer automatically fills the holes in this sketch such that it satisfies a given specification. We refactor one line of program $\mathcal{I}$ at each time. Our first step is to use the ranked list of APIs to create a program sketch where the parameters are unknown. For instance, consider the first layer from the motivating example that shows the network for an autoencoder using TensorFlow:

```
tf.keras.layers.Conv2D(filters=32, kernel_size=3, strides=(2, 2))
```

A possible sketch for this call using PyTorch is:

```
torch.nn.Conv2d(#1, #2, (#3,#4), stride=(#5, #6), padding=(#7, #8))
```

Where holes `#i` have to be filled with a specific value for the APIs to be equivalent. This approach works for one-to-one mappings but would not support common one-to-many mappings where the parameters often need to be transformed before being used in the new API. This is the case of the previous API where a reshaping operation must be performed before calling the PyTorch API. To support this common behavior, we include in our program sketch one API from the target library $\mathcal{T}$ and common reshaping APIs (e.g., permute).

The sketch that corresponds to the refactoring solution of the `Conv2D` API from TensorFlow uses a reshaping API before calling the `Conv2d` API from PyTorch:

```
lambda x: x.permute(#9, #10, #11, #12)
torch.nn.Conv2d(#1, #2, (#3, #4), stride=(#5, #6), padding=(#7, #8))
```

Using Occam's razor principle, our program synthesizer enumerates program sketches of size 1 and iteratively increases the size of the synthesized program up to a specified limit.

**Program Enumeration.** For each program sketch $\mathcal{P}$, our program synthesizer enumerates all possible completions for each hole. Since each hole has a given type, we only want to enumerate well-typed programs. We encode the enumeration of well-typed programs into a Satisfiability Modulo Theories (SMT) problem using a combination of Boolean logic and Linear Integer Arithmetic (LIA). This encoding is similar to other approaches that use SMT-based enumeration for program synthesis [128] and encodes the following properties:

- Each hole contains exactly one parameter;
- Each hole only contains parameters of the correct type.

A satisfying assignment to the SMT formula can be translated into a complete program. The types for each hole can be determined by extracting this information from documentation, by performing static analysis, or by having this information manually annotated in the APIs. The available parameters and their respective types can be extracted automatically from the parameters used in the $k$-th line of program $\mathcal{I}$ and by any default parameters that can be used in the API from $\mathcal{T}$ that appears in the program sketch $\mathcal{P}$. For instance, for the `Conv2d` example presented in this section, we consider as possible values for the holes, the values that appear in the existing code (32, 3, 2) and default values for integer parameters (-1, 0, 1, 2, 3) that are automatically extracted from documentation.

Encoding the enumeration of well-typed programs in SMT has the advantage of making it easier to add additional logical constraints that can prune the search space.

### 3.2.4 Documentating Metadata to guide Synthesis

As we described in Section 3.2.1, API documentation often provides additional useful information about parameters to function calls, including type and default values. For each considered API call, we scrape/process

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

**Figure 3.4:** Relationship between the parameters of Conv2d API described in PyTorch documentation [120].

the associated documentation to extract these properties and encode them as SMT constraints to further limit the synthesizer search space.

Additionally, some APIs have complex relationships between parameters which if encoded into SMT may reduce the search space considerably. For instance, Figure 3.4 shows the relationship between the different parameters for the Conv2d API described in PyTorch documentation. For APIs with these kinds of shape constraints, we can encode these relationships into SMT to further prune the number of feasible completions. When we use these relationships in our experiments, we encode them manually (a one-time cost for an actual SOAR user or API maintainer), but we observe that in many cases they could be automatically extracted from documentation.

### 3.2.5 Error Message Understanding

Besides meta-data, we can also guide the refactoring engine using the error messages provided by the Python interpreter. The idea is to give feedback to the program synthesis engine during the refactoring process.

We use a simple natural language processing approach to extract data from compiler error messages. Specifically, we extract hyponymy relations and use Word2vec [129] to understand run-time error messages. We transform error messages into constraints for our synthesizer. Figure 3.5 illustrates the process.

**Step 1: Extract hyponymy relation candidates from error messages.** We perform an automatic extraction of customized hyponyms on each error message. Hyponyms are specific lexical relations that are expressed in well-known ways [130]. In encoding a set of lexico-syntactic patterns that are easily recognizable (i.e., hyponyms), we avoid the necessity for semantic extraction of a wide-range of error message text. We then use the collected hyponyms to map the error message to a single faulty parameter, and output a SMT constraint based on the faulty parameter.

We use four manually crafted lexico-syntatic patterns to identify hyponyms using *noun-phrases* (NP) and regular expressions frequently appearing in machine learning API error messages.

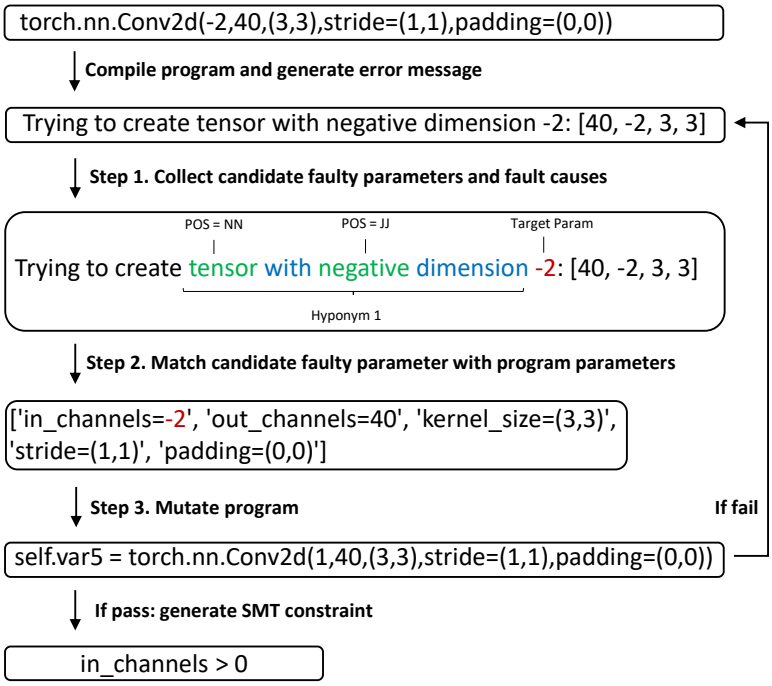



**Figure 3.5:** Example error message to SMT constraint pipeline using hyponym 1.

**Step 2: Identify candidate faulty parameters and constraints.** Step 2 uses different keywords based on the result of step 1 to identify the faulty parameter. As shown in Figure 3.5, an error message with hyponym 1 is likely to have the POS=JJ word as a parameter constraint (i.e., word "*negative*"). Based on the fault cause candidate, we then store all negative numbers as candidate faulty parameters (e.g., [40, -2, 3, 3] has -2 as the only faulty parameter). We then vectorize the candidate faulty parameter name (i.e.,-2) and find the program parameter name with the closest vectorized distance. As shown in Figure 3.5, the parameter "$in\_channels = -2$" has the nearest vectorized distance to the candidate faulty parameter -2. Based on the fault cause, we generate a candidate constraint. The example error message in Figure 3.5 has only one candidate constraint: "$in\_channels >= 0$".

**Step 3: Mutate program.** To validate the candidate faulty parameters and constraints, we mutate each faulty parameter according to each faulty parameter and constraints pair. We then re-compile the program for each mutation. If the error message remains the same, we discard the faulty parameter and constraint pair as a candidate. If the program passes, or if the error message changes, we store the faulty parameter and constraint pair as an SMT constraint. As shown in Figure 3.5, the API call mutator mutates the second parameter ("$in\_channels = -2$") to a non-negative number. The mutator first attempts "$in\_channels = 0$" and it encounters a different error message. From the new error message, we mutate this parameter to "$in\_channels = 1$" and observe no further errors. Therefore, we refine our previous constraint to be "$in\_channels > 0$", and store it as the final SMT constraint for the program in Figure 3.5.

## 3.3 Evaluation

We evaluate our approach by answering the following research questions:

**RQ1.** How effective is SOAR at migrating neural network programs between different libraries?

**RQ2.** How effective is API documentation to establish mappings? .

**RQ3.** How effective is API meta-data in guiding the refactoring process?

**RQ4.** How useful are error messages in guiding the refactoring proccess?

**RQ5.** Is SOAR generalizable to domains besides deep learning library migration?

### 3.3.1 Benchmarks and experimental setup

We collected 20 benchmarks for each of the two migration tasks. In particular, for the `TensorFlow` to `PyTorch` task, we gathered 20 neural network programs from `TensorFlow` tutorials [131], existing models implemented with `TensorFlow` [132] or its model zoo [133]. This set of benchmarks includes: Autoencoders for image and textual data, classic feed-forward image classification networks (i.e., the VGG family, AlexNet, LeNet, etc), convolutional network for text, among others. The average number of layers in our benchmark set is 11.80 ± 11.52, whereas the median is 8. Our largest benchmark is the VGG19 network which contains 44 layers.

For the domain of table transformations, we collected 20 benchmarks from Kaggle [134], a popular website for data science. The programs in the benchmark set have an average of 3.05 ± 1.07 lines of code, and a median of 3 lines. Although the programs considered for this task are relatively small compared to the deep learning benchmarks, they are still relevant for data wrangling tasks as shown by previous program synthesis approaches [135].

All results presented in this section were obtained using an Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz, with 64GB of RAM, running Debian GNU/Linux 10, and a time limit of 3600 seconds. To evaluate the impact of each component in SOAR, we run four versions of the tool. SOAR with TF-IDF (SOAR w/ TF-IDF) and SOAR with tfidf-GloVe (SOAR w/ Tfidf-GloVe) to evaluate the impact of API representation learning methods. SOAR without specification constraints (SOAR w/o Specs.) and SOAR without error message understanding (SOAR w/o Err. Msg.) to evaluate the impact of these components on the performance of SOAR.

### 3.3.2 Implementation

The SOAR implementation integrates several technologies. Scrapy [136], a Python web-scraping framework, is used to collect documentation for the four libraries in our experiments. To enumerate programs in the synthesis step, we use the Z3 SMT solver [137]. For each target program call parameter, we extract an answer for the four parameter questions in Section 3.2.1 and generate corresponding SMT constraints. In both API matching model and the error message understanding model, the GloVe word embeddings [122] are used

**Table 3.1:** Execution time for the deep learning library migration task in each of the 20 benchmarks.

| | SOAR | SOAR w/o Specs. | SOAR w/o Err. Msg. |
|---|---|---|---|
| conv_pool_softmax(4L) | **1.60** | 23.02 | 14.35 |
| img_classifier(8L) | **12.82** | 336.00 | 65.66 |
| three_linear(3L) | 3.18 | **2.34** | 21.07 |
| embed_conv1d_linear(5L) | **5.27** | 123.85 | 16.90 |
| word_autoencoder(3L) | 1.81 | **1.46** | 2.64 |
| gan_discriminator(8L) | **12.80** | timeout | 252.20 |
| two_conv(4L) | 16.69 | timeout | **15.09** |
| img_autoencoder(11L) | **160.97** | 391.09 | 487.54 |
| alexnet(20L) | 425.22 | timeout | **66.13** |
| gan_generator(9L) | **412.47** | timeout | timeout |
| lenet(13L) | **280.91** | timeout | timeout |
| tutorial(10L) | **6.04** | timeout | 58.29 |
| conv_for_text(11L) | **9.04** | timeout | 32.29 |
| vgg11(28L) | **40.83** | timeout | 132.67 |
| vgg16(38L) | **82.05** | timeout | 139.27 |
| vgg19(44L) | **83.99** | timeout | 189.90 |
| densenet_main1(5L) | timeout | timeout | timeout |
| densenet_main2(3L) | timeout | timeout | timeout |
| densenet_conv_block(6L) | timeout | timeout | timeout |
| densenet_trans_block(3L) | timeout | timeout | timeout |

as an off-the-shelf representation of words. For the four libraries appearing in our two evaluation migration tasks, we use `TensorFlow` 2.0.0, `PyTorch` 1.4.0, `dplyr` 1.0.1 (with R 4.0.0) and `pandas` 1.0.1, though our proposed method and associated implementation do not rely on specific versions.

### 3.3.3 Results

#### 3.3.3.A RQ1: Overall SOAR effectiveness

Table 3.1 shows how long it takes to migrate each of the deep learning models from `TensorFlow` to `PyTorch`, using the various approaches. Our best approach (shown as SOAR) successfully migrates 16 of the 20 DL models with a mean run-time of $97.23 \pm 141.58$ seconds, and a median of 14.76 seconds. The average number of lines in the 16 benchmarks that we successfully migrate is $13.6 \pm 12.14$, whereas the average number of lines in the output programs is $18.56 \pm 16.40$. The reason the number of synthesized lines is higher than those in the original benchmarks is that we frequently do one-to-many mappings. In fact, 15 out of the 16 require at least one mapping that is one-to-many. In the 16 benchmarks, SOAR tests on average $4414.18 \pm 5676$ refactor candidates (i.e. program fragments tested for each mapping), and it needs to test a median 2111 candidates before migrating each benchmark. The reason 4 benchmarks timeout is that in each of these benchmarks there is at least one API in the benchmark that has a poor ranking (i.e., not in the top 200).

**Table 3.2:** Execution time and average API ranking for each of the 20 benchmarks using TF-IDF and GloVe models.

| | SOAR w/ TF-IDF | | SOAR w/ Tfidf-GloVe | |
|---|---|---|---|---|
| | Time(s) | Avg. Ranking | Time(s) | Avg. Ranking |
| conv_pool_softmax(4L) | 1.60 | 1.0 | **1.56** | 1.0 |
| img_classifier(8L) | **12.82** | 2.8 | 31.04 | 2.8 |
| three_linear(3L) | 3.18 | 8.0 | 7.70 | 8.0 |
| embed_conv1d_linear(5L) | **5.27** | 2.4 | 7.75 | 2.4 |
| word_autoencoder(3L) | 1.81 | 1.0 | **1.52** | 1.0 |
| gan_discriminator(8L) | **12.80** | 2.8 | 37.01 | 2.8 |
| two_conv(4L) | 16.69 | 1.0 | **13.75** | 1.0 |
| img_autoencoder(11L) | **160.97** | 1.9 | 166.34 | 2.0 |
| alexnet(20L) | **425.22** | 2.1 | 428.42 | 2.1 |
| gan_generator(9L) | **412.47** | 2.0 | 1892.86 | 2.0 |
| lenet(13L) | **280.91** | **4.3** | timeout | 89.1 |
| tutorial(10L) | **6.04** | **2.3** | 21.31 | 2.4 |
| conv_for_text(11L) | **9.04** | 2.3 | 14.08 | 2.3 |
| vgg11(28L) | **40.83** | 1.8 | 73.92 | 1.8 |
| vgg16(38L) | **82.05** | 1.6 | 114.41 | 1.6 |
| vgg19(44L) | **83.99** | 1.5 | 114.98 | 1.5 |
| densenet_main1(5L) | timeout | **172.8** | timeout | 285.4 |
| densenet_main2(3L) | timeout | **16.0** | timeout | 387.5 |
| densenet_conv_block(6L) | timeout | **293.3** | timeout | 612.7 |
| densenet_trans_block(3L) | timeout | **291.0** | timeout | 480.0 |

#### 3.3.3.B RQ2: How effective is API documentation to establish mappings?

In Table 3.2, we show results of `SOAR` using different API representation learning methods, namely TF-IDF and TFIDF-GloVe, as described in Section 3.2. We can see that for these tasks of `TensorFlow` to `PyTorch` migration, using TF-IDF-based API matching model works better than adding pretrained GloVe embeddings. We believe this is because similar APIs are often named with same words(e.g., `Conv2DTranspose` vs. `ConvTranspose2d`) or even identical name (e.g., the APIs of creating a Rectified Linear Unit are both named as `ReLU(...)`), for `TensorFlow` and `PyTorch`. Thus simple word matching method like TF-IDF is suffice for API matching purposes.

Another interesting result worth noticing is that although the synthesis time differs for the two approaches, the average rankings are quite similar for most of the benchmarks. The reason is that despite the average rankings of correct target APIs being similar, the incorrect APIs ranked by the model before the correct one is different, and the time it takes to rule out those incorrect APIs varies greatly, determined largely by the number of parameters required for that API.

#### 3.3.3.C RQ3: How effective is API meta-data in guiding the refactoring process?

In Table 3.1, we also show the impact of specification constraints that describe the relationship between different parameters of a given API (see Section 3.2.3 for details). Even though, we only have these complex

specifications for the 7 most common APIs, the impact on performance is significant. Without these specification we can only solve 6 out of 20 benchmarks. Relating the arguments of the APIs helps SOAR to significantly reduce the number of argument combinations that it needs to enumerate.

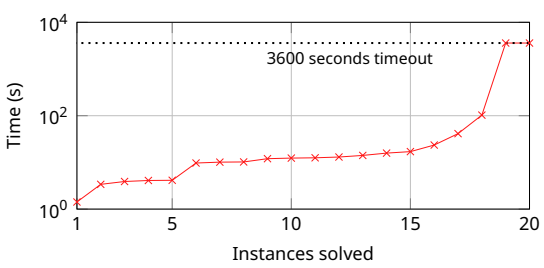
#### 3.3.3.D RQ4: How useful are error messages in guiding the refactoring proccess?

As shown in Table 3.1, SOAR performs significantly better when using the error message understanding model. We can observe that without this component, two of the benchmarks that SOAR could solve would timeout at the 1 hour mark. For the 14 benchmarks it still manages to solve, the synthesis time increases on average 4.66×. The number of performed evaluations also increase substantially for each benchmark. For the 16 benchmarks that SOAR successfully migrates, we evaluate an average of $43319.63 \pm 61259.62$ refactor candidates without the error message understanding model. This corresponds to a 9.81× increase in the number of necessary evaluations when compared to the full SOAR method. In summary, we can significantly reduce the search space by interpreting error messages.

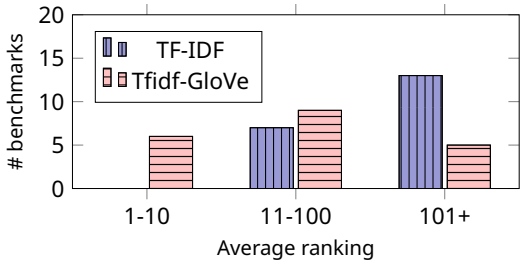
#### 3.3.3.E RQ5: Generalizability of results

Our experiments so far concern deep learning library migration in Python. To study the generality of our proposed SOAR, we applied SOAR to another task of migrating from `dplyr`, a data manipulation package for R, to `pandas`, a Python library with similar functionality. Fig. 3.6b shows how the two API matching methods perform in this domain. While with Tfidf-GloVe, 30% of the correct APIs are ranked among the top 5, saving lots of evaluations for the synthesizer, none of the correct APIs are ranked by the TF-IDF-based matcher as its first 5 choices. Worse, nearly half of those are ranked above 100, making the synthesis time almost prohibitively long. We believe this is because the lexical overlap between the names of similar APIs in those two libraries is much smaller compared to the deep learning migration task. For example, dplyr's `arrange` and panda's `sort_values` provide the same functionality (they both sort the rows by a given column), but the function names are different. In this way, Tfidf-GloVe can take advantage of the pretrained embeddings to explore the similarities between APIs beyond simple TF-IDF matching.

In Figure 3.6a, we show the time it takes to migrate each of the 20 benchmarks with a timeout of 3600 seconds when using word embeddings. We solve 18 out of 20 collected benchmarks in under 102.5 seconds. The average run time for 18 benchmarks is $17.31 \pm 22.59$ seconds and a median of 12.19 seconds. Note that for this task we did not consider error messages, nor specifications since we wanted to test how a basic version of SOAR would behave in a new domain. Moreover, for this domain, all the refactored benchmarks only used one-to-one mappings since no additional reshaping was needed before invoking `pandas` APIs. Even with these conditions, we show that we are able to successfully refactor code for a new domain across different languages.




**(a)** Execution time for each benchmark of the dplyr-to-pandas task with a timeout of 3600 seconds.




**(b)** Average ranking of the APIs for each of the 20 dplyr-to-pandas benchmarks.

**Figure 3.6:** Comparative results of dplyr-to-pandas task.

## 3.4 Discussion and threats to validity

Overall, we focus our design and evaluation on deep learning and data science libraries. These libraries have properties that render them well-suited to our task in terms of common programming paradigms, and norms, such as in the API documentation. However, we believe this is also a particularly useful domain to support, given the field's popularity and how quickly it moves, how often new libraries are released or updated, as well as the wide variety of skill sets and backgrounds present in the developers who write data science or deep learning code. Automation of migration and refactoring in this domain is very minimal, and we design SOAR as a step towards better tool support for this diverse and highly active developer population.

Next, we discuss the main limitations of our method and possible challenges for extending SOAR's ability to refactor new APIs, even potentially beyond the domain of data science.

**Benchmarks.** Our evaluation of SOAR uses benchmarks from well-known deep learning tutorials and architectures. However, they are all feed-forward networks, effectively sequences of API calls where the output of the current layer is the input of the next layer. There may be more applications that share this feature, but support for more complex structure is likely necessary to adapt to other domains.

Additionally, and naturally, the APIs in the benchmarks we collected may be biased and not reflect the set of APIs developers actually use. To assess this risk, we checked the degree to which the APIs used in our benchmarks appear to be widely used on other open-source repositories on GitHub. To do this, we collected the top 1015 starred repositories that have `TensorFlow` as a topic tag, which contains over 8 million lines of code and over 500K `TensorFlow` API calls. We found that 76% of the 1000+ repositories use API calls included in our benchmarks at least once, which validates some representativeness of our collected benchmarks.

**Automatic testability.** One benefit of the data science/scientific computing domain is that much of the input, output, and underlying methods are typically well-defined. As a result, it's particularly easy to test and verify the correctness of individually migrated calls, which can be processed in sequence. There may be other types of libraries that share these types of characteristics, like string manipulation or image processing libraries, whose intermediate outputs are strings/images. We also assume user-provided tests. Given the migration

task, it is reasonable to assume the user has tests (the code must be sufficiently mature to justify migrating, after all), but a more general solution might benefit from automatically generating tests, which would both alleviate the input burden on the user and, potentially, reduce the risks of overfitting. In our current implementation, we moreover use the provided tests to construct smaller test cases for each mapping. This is particularly easy in this domain, because data science and deep learning API calls are often functional in their paradigm. Adapting the technique to other paradigms would require more complex test slicing or generation to support synthesis.

**Correctness.** Since we evaluate our migration tasks using test cases, it is always possible for our approach to overfit to said test cases. However, this threat can be mitigated if the user provides a sufficiently robust test suite that provides enough coverage.

Additionally, code written to different APIs may be functionally equivalent, but demonstrate different performance characteristics, which we do not evaluate. However, this fact is one reason users might find SOAR useful in the first place: a desire to migrate code from one library to another that is more performant for the given use case.

**Error message understanding.** The error message understanding model is built on four domain specific lexico-syntatic patterns, which we identify as hyponyms when they appear in an error message. We propose the hyponyms based on the specific syntax of DL API error messages, thus take non-trivial human effort to make it generalize to error messages that appear when calling APIs from libraries of other domains. However, we believe the idea of program mutation (Step 3 of Fig. 3.5) is still widely applicable for the purpose of generating SMT constraints when dealing with error messages.

**Synthesis.** Our approach supports one-to-many mappings but it restricts the mapping to one API of the target library and one or more reshaping APIs. However, this could be extended to include many APIs of the target library at the cost of slower synthesis times. An additional challenge is to support many-to-one or many-to-many mappings since this would require extending our synthesis algorithm. However, even with the current limitations, our experimental results show that the current approach can solve a diverse number of benchmarks.

## 3.5 Key Takeaways and Contributions.

In this chapter, we demonstrate that API documentation can serve as a proxy to establish API mappings for migration. We used these mappings to guide a Synthesis approach for API Refactoring (SOAR). SOAR uses a generate-and-test strategy, as computing mappings alone is insufficient for completing a migration; API arguments and glue code also need to be mapped. Moreover, we cannot blindly trust the mappings derived from the documentation; indeed, the correct mappings are sometimes not correct (i.e., corresponding APIs may be apart in the embedding space, meaning they might be closer to other, non-semantically equivalent APIs).

For these reasons, to complete migrations, we test the API mappings with multiple different combinations of arguments using synthesis. We determine that a particular API migration is complete if the synthesized code produces the same output on a set of auto-generated inputs. During the synthesis process, the interpreter also outputs warnings or errors due to API usage. We leverage this information with a simple error message understanding mode, which we use to prune the search space.

Overall, our approach successfully migrates API calls within reasonable time frames, particularly for small programs (under 100 lines) where it is feasible to compare objects across implementations. Since we synthesize code directly rather than generating migration scripts, this method is not generally applicable for large-scale refactorings of code bases comprising millions of lines of code. We tackle this limitation in Chapter 6, where we show how to synthesize migration scripts for library migrations.

# 4

# Mining API Refactoring Rules from the API Development Process

## Contents



In this section, I discuss my completed and published work titled "Mining Effective Lightweight Transformations from Pull Requests" (MELT) [45]. In this work, we leverage the API development process from open-source libraries (i.e., pull requests) to mine rules for fixing breaking changes between library versions. One core idea of MELT is to identify pull requests (PRs) between library releases that break existing APIs. Upon identifying these PRs, we extract data from a variety of sources. First, we identify code changes to test cases that developers made after breaking the API. Our key observation is that if a library is well-tested, developers need to update test cases when they break an API; otherwise, the tests would fail. This data allows us to mine

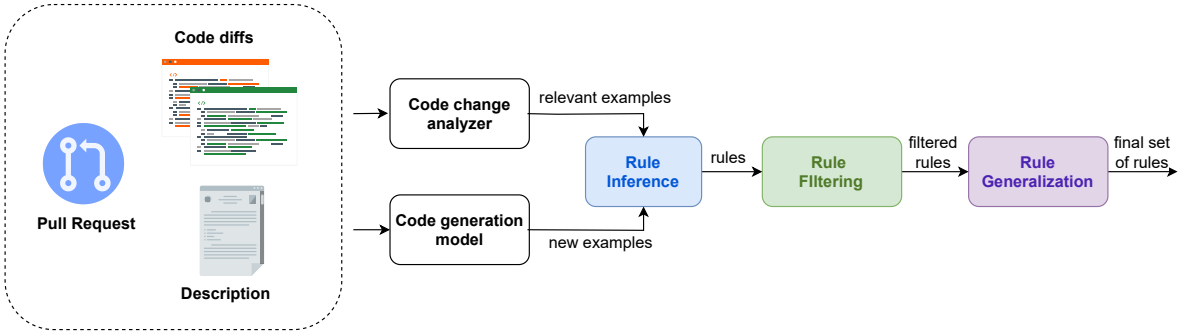



**Figure 4.1:** MELT takes as input a pull request (PR) and outputs a set of rules. The PR is processed in two ways: (1) the Code change analyzer identifies relevant code changes; (2) the Code generation model generates additional code examples. Rules are inferred from the code changes and examples, then filtered and generalized.

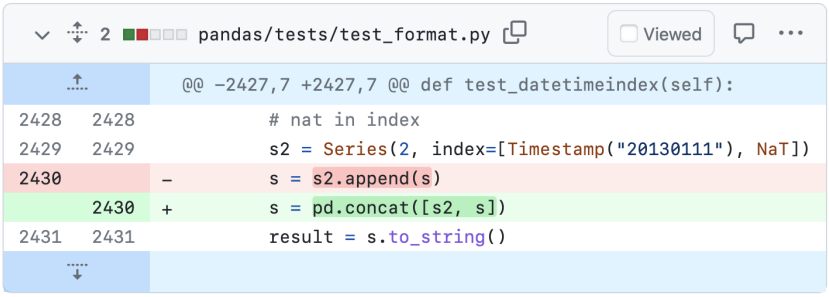



**Figure 4.2:** Code change in pull request #44539 [138] from the `pandas-dev/pandas` repository.

rules directly from the library instead of relying on downstream clients like the majority of state-of-the-art methods. The migrations themselves are expressed in the `comby` language for broader applicability. This means the synthesis only needs to happen once.

## 4.1 Motivating Example

Figure 4.1 provides a high-level overview of MELT and its main components. We delve into the specifics of each component in Section 4.2.

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 4.2 shows an example code change from a pull request [138] submitted to `pandas` [139] that deprecates two popular APIs: `DataFrame.append` and `Series.append`.[1] MELT identifies code changes, such as the one shown in Figure 4.2, within the pull request using its *Code Change Analyzer* (Section 4.2.1) and inputs them into the *Rule Inference* algorithm (Section 4.2.3.A) to generate rules. The top portion of Table 4.1 shows two of the rules MELT infers from the code changes for this specific pull request.

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

**Table 4.1:** 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 and :[[s2]].type == Series` | `pd.concat([:[[s2]], :[[s1]]])` |
| | `:[[df]].append(:[[s]])` where `:[[df]].type == DataFrame and :[[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])` |

The rules in Table 4.1 are expressed in comby's domain specific language [140]. 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 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 useful when working with common API names such as `append` and `concat` (both are part of Python's stdlib).

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 4.3 shows an informative comment left by a developer in a code file when deprecating namespace `scipy`'s [141] 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 4.2.2). Figure 4.4 shows a simplified version of code `GPT-4` [39] (a state-of-the-art model) generates from the pull request in Figure 4.3. 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 4.1. 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

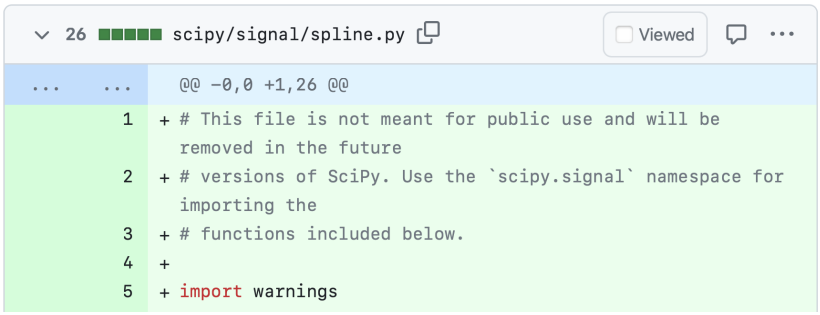



**Figure 4.3:** Pull Request #14419 [142] from `scipy/scipy`. This pull request was part of SciPy 1.8, released in Feb 2022.

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

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

**(a)** Old and new usage of `cspline2d`.

```python
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))
```

**(b)** Test case for the transition.

**Figure 4.4:** Code generated by GPT−4 showcasing how to transition from the old `cspline2d` usage and a test case.

arguments, including keyword arguments with default values.

To guard against overly-specific rules, MELT applies *Rule Generalization* (Section 4.2.3.C). For example, the template holes `:[[x]]` and `:[y]` in the rule in the first row of the bottom of Table 4.1 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 4.1). The revised rule uses a more permissive match template using `:[args]`, which can match any number of function arguments.

## 4.2 MELT's Approach

In this section, we provide a brief overview of MELT's approach.

### 4.2.1 Extracting Code Examples from Diffs in Pull Requests

MELT's input is a pull request $\mathcal{P}$, which contains both natural language descriptions and a set of code 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 4.2, 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.[2] 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.

### 4.2.2 Generating Examples for Mining using Natural Language

As illustrated in Section 4.1, 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.

We developed prompts and conducted experiments with `GPT-4 8K` [39], which is well-versed in our target libraries' code, to process PR information (code diffs, title, description, discussion). For each PR, we asked the model to generate: 1. transition examples, and 2. test cases asserting that the old behavior was the same as the new one. Full prompts and algorithmic description can be found in the paper. However, the key idea is to check if automatically generated tests are correct according to the automatically generated test suite. To make sure the test suite is not spurious, we sample multiple tests. If any test fails, the example is not considered for mining.

### 4.2.3 Rule Mining from Examples

`MELT` uses the `comby` language [23] and toolset [140] to express refactoring match-replace rules. We introduced some elements of the language in Section 4.1, with examples of `comby`'s syntax-driven match and rewrite templates. Formally, a rewrite rule in `comby` is of the form $\texttt{M} \rightarrow \texttt{R where } c_1, c_2, \ldots, c_n$, where

[2] 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.

**(a)** Code before migration

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

**(b)** Code after migration

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

**Figure 4.5:** Example code change from PR #43242 [144] in `pandas`

`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 [140] 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 ASTs.

#### 4.2.3.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 [143] (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 4.5, 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]] = :[[i]].read_csv( [[d]],
    compression=:[e], encoding=:[f], →
    index_col=[[g]], squeeze=:[[h]])
```

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

```
:[[a]] = :[[i]].read_csv(:[[d]],
    compression=:[e], encoding=:[f],
    index_col=:[[g]]).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 [143, 145], 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 [146] type inference language server, making it available for client usage.

#### 4.2.3.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:

**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.

**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.

#### 4.2.3.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. MELT removes common context in the source and match templates unrelated to the API. For our example, it unwraps the assignment statement and simply keeps the API call. MELT also uses InferRules [143] algorithm to find mappings between call nodes in the match and rewrite template, and generalizes common arguments. If there are multiple consecutive arguments between the match and rewrite template of the call node, we replace the arguments with a generic template variable `:[args]`. For our running example, the final rule is:

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

Generalization is crucial to ensuring broader rule applicability. The paper describes the generalization algorithm in more detail.

## 4.3 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?

### 4.3.1 Experimental Setup

#### 4.3.1.A 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 [147], and (2) there exists a gap in migration tools for Python [27]. We implemented rule inference using the Python abstract syntax tree (AST) module. `InferRules` [143] 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` [39]. We extended `comby` to support Language Server Protocol (LSP)-based type inference over match templates [148] with `Jedi` [146], a state-of-the-art static analysis tool. MELT's source code, data, and logs used for the evaluation are available at Zenodo [149].

**Table 4.2: 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 | PRs with Correct Rules | Mined Rules Total | Mined Rules 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 %) |

#### 4.3.1.B 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.

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) [150]. Due to the high agreement, the first author labeled the remaining rules to cover all research questions.

### 4.3.2 Results

#### 4.3.2.A RQ1: Mining Rules from Code Examples in PRs

Table 4.2 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 [151] 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 4.2 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 [152], 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 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 [153] 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.

#### 4.3.2.B RQ2: Mining Rules from Autogenerated 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 5 times to generate transition examples. We then followed up with the model to ask for test cases for each sample (in total 10 requests per PR).

The left side of Table 4.3 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 4.3.2.A), 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 4.3 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]))`.[3] This rule is derived from pandas pull request number #49362 [154]. The release notes for the PR state:

[3] Template variables are omitted for brevity.

**Table 4.3:** 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 |

*"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.

#### 4.3.2.C 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 [155],[4] 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 50 random repositories for each rule, and ran the generalized and non-generalized rules on these repositories, counting matches.

Table 4.4 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.

#### 4.3.2.D 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

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

**Table 4.4:** 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 |

and running client projects 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 [149]. 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 4.5 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.

**Table 4.5: 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 |

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.

## 4.4 Discussion

In this section, we address the main limitations of our approach.

### 4.4.1 Limitations and threats

#### 4.4.1.A 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 [151]. 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. However, it is also important to note that we are limited by the expressiveness of the language in which we represent the changes.

#### 4.4.1.B 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.

#### 4.4.1.C 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. This limitation stems from the expressiveness that `comby` language provides, rather than `MELT`'s approach. Regardless, `MELT` can return both kinds of rules to the user (i.e., specific or generalized), 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.

### 4.4.2 Comparison against prior work

Few API migration tools target Python, challenging direct comparison to prior work. `MELT` adapts its inference algorithm from `InferRules` [143], 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` [145], builds on `InferRules` using `comby` as an intermediate representation. `PyEvolve` focuses

on general refactoring, and adapts rules to different control variants, requiring more complex 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 [18, 21] 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 2.4, 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 [19] (focusing on Android API migration), or APIFix [18] (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 [21]. 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 [156]), 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.

## 4.5 Key Takeaways and Contributions

In this chapter, we demonstrated that the library development process itself can be used to mine migration rules to address simple API breaking changes. It is not necessary to rely on commit data from client projects that have already undergone migration. This assumption had previously limited the migration approaches, especially immediately after a library version is released. By integrating the mining process directly with the library workflow in CI, migration rules can be provided to clients immediately after the source code is updated.

MELT represents migration rules in comby, a lightweight match-replace language for large-scale code transformation. Since mining can result in very specific rules, we developed a generalization procedure to

increase rule applicability. However, this approach comes with trade-offs: some rules are overly generalized and thus may produce incorrect code, while others are too specific. Although the work was successful, some migrations could not be expressed, primarily due to limitations of the language we used for the synthesis (`comby`). In the next, we further explain and exemplify these problems, and introduce a more suitable language to express complex API migrations.

# 5

# A Language for Automating Logically Related Changes

## Contents



In this chapter, I present my published and completed work on a new code transformation language. This language is designed to allow expressing flow, dependencies, and the composition of match-replace rules. This effort is driven by the observation that most code changes (including library migrations) tend to be cascading and interconnected; yet, modern languages for code transformations do not inherently offer support for expressing sequences of changes. Indeed, we faced this problem in Chapter 4, where finding

a balance between overgeneralization and overly specific rules was a persistent challenge, largely due to expressiveness limitations stemming from the `comby` language.

## 5.1 Motivation

Library migrations usually result in a web of cascading and interdependent code changes that span and propagate across multiple files or repositories [157]. For example, consider the library migration from Figure 5.1, where the goal is to replace `log4j` (a logging library for `Java`) with an alternative `slf4j`. We divide the migration in four steps:

1. **Migrating Imports**: Replace the import statement for the `Logger` type, `org.apache.log4j.Logger`, with its `slf4j` equivalent, `org.slf4j.Logger`.
2. **Migrating Instantiation**: In contrast to `log4j`, in `slf4j` logger objects are instantiated using a factory method pattern (implemented in the `slf4j.LoggerFactory` class). Therefore, it is necessary to replace the `Logger.getLogger` call with `LoggerFactory.getLogger`.
3. **Adding Missing Import**: Since `getLogger` is a method of a different class (`slf4j.LoggerFactory`), it is also necessary to include the appropriate import statement for this class.
4. **Migrating Associated Method Calls**: Migrate all method calls associated with the logger to their `slf4j` equivalents. In this example, it is only necessary to update the `logger.info` usage.

While it might be feasible to express these migration steps as individual transformation rules (for example using `comby` [140]), such a strategy is suboptimal for multiple reasons.

**Runtime Performance.** Applying transformation rules indiscriminately to an entire code base, regardless of the file's relevance to the type or object in question, can lead to significant overhead and result in slow and resource-intensive tool executions. For example, suppose our goal is to migrate a code base from `log4j` to `slf4j` as described above. Here, we notice that despite there being multiple steps in the migration, only files with logger objects need to be touched. One proxy for detecting such files is by looking up an import statement, as imports typically indicate usage. This means that our migration scripts can be designed to only attempt to migrate a file (and execute the rules corresponding to the migration), if an import statement to `log4j` is present. This is particularly relevant when dealing with large-scale migrations across codebases with millions of lines of code.

**Accuracy and Precision.** Match-replace rules typically provide limited control over which code should be transformed. For example, the import statement in (Line 2, Step 3) is necessary only if the `getLogger` API is migrated within the file and the import is not already present. Additionally, the migration of the info API (Line 8, Step 4) should only affect the logger object that was migrated in (Line 5, Step 2). Calls to other objects with an `info` API from another object should not be affected. Writing such constraints in


**Figure 5.1:** Migration steps to move from two popular logging libraries in `java`: `log4j` and `slf4j`.

the `comby` language would require additional scripting. An attempt at writing this rule in `comby` could be `:[x].info(:[args])-> :[x].info("{}", :[args])`. However, such a rule would apply across the entire codebase and not just to the previously migrated logger object. On the other hand, substituting `logger` for `:[x]` would overly specialize the rule to this example: `logger.info(:[args])-> logger.info("{}", :[args])`. This is particularly challenging to address when migrating common API names like `append` and `concat` as discussed in Chapter 4.

**Expressiveness.** Current techniques cannot express complex automations as match-replace rules. Generally, to perform complex automations, developers have to rely on imperative code transformation frameworks instead. These frameworks (e.g., [74, 91]) provide APIs for manipulating code at the AST level, allowing for arbitrary code transformations. The APIs let users control where, when, and how code should be rewritten based on context, symbol information, and more complex analyses. However, imperative frameworks present a twofold problem. First, imperative frameworks are typically monuments of engineering, demanding signif-

icant time and effort to learn [158]. Second, the frameworks are typically language-specific, as they rely on compiler and build infrastructure to be able to get meaningful information from the code. This results in significant burdens as multiple developer experts are necessary for automating the same task in different languages [90].

## 5.2 Preliminaries

### 5.2.1 Existing Code Transformation Languages

Frameworks for automating code transformation vary widely. At one end of the spectrum, *lightweight techniques* [23, 75, 159] offer declarative languages to rewrite code with simple match-replace rules. The key advantage of lightweight techniques is language agnosticism, which stems from these techniques being independent of the underlying compiler infrastructure. Moreover, match-replace rules are often syntactically close to the target language, making them easy to write and use [23]. However, lightweight techniques are often limited to atomic context-free code changes, lacking support for tasks requiring cascading and interdependent code changes. On the other hand, *imperative frameworks* [160] for AST-level manipulation allow for arbitrary code transformations. As discussed earlier, imperative frameworks provide infrastructure and finer control over where, when, and how code should be rewritten based in analyses. These frameworks may not be the best fit for learning migration scripts due to their large APIs, and their imperative nature. Synthesizing arbitrary imperative programs is undecidable.

### 5.2.2 A novel Lightweight Language for Cascading Transformations

To address the limitations of existing lightweight match-replace tools (as described in 5.1), we have developed a new language called PolyglotPiranha. At a high level, PolyglotPiranha allows for sequencing of rule applications as well as propagating information across rules using a *directed graph of match-replace rules*.

Programs in PolyglotPiranha are graphs, where the nodes represent individual transformation rules (expressed in a code transformation language of choice), and the edges determine the order for applying these rules. Each edge is also associated with a label that defines the scope within which the target rule is applied with respect to the source rule. For example, an edge $\mathcal{R}_1 \xrightarrow{\text{class}} \mathcal{R}_2$ reads as, "Apply rule $\mathcal{R}_1$ and then apply rule $\mathcal{R}_2$ within the enclosing class where $\mathcal{R}_1$ was applied." The ability to cascade transformations using match-replace rules makes it an ideal candidate for writing API refactoring rules.

Note that the goal of this research is not to design a new matching syntax language per se, but rather a meta-language that allows for the combination and interleaving of existing match-replace languages, as well as provide infrastructure to combine and sequence them. We build our approach on top of this idea of graph of match replace-rules. In PolyglotPiranha:

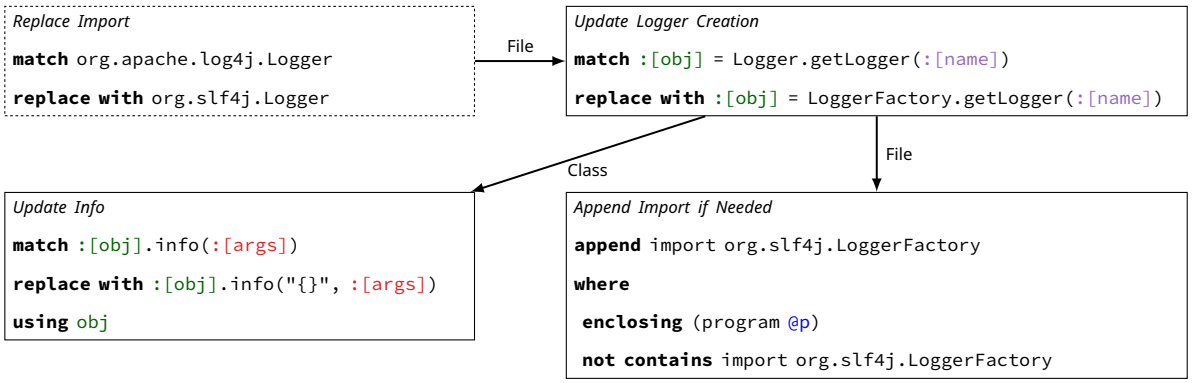



**Figure 5.2:** Program in the DSL to described the migration for Figure 5.1. The dashed line represent a seed rule, i.e., the rule that triggers the migration. Each edge is annotated with a scope. The scope determines where the target rule will be applied with respect to the source.

1. We aim to allow users to express match-replace rules in *any* source code matching language. We provide alternative syntaxes, catering to different users and different needs. The current implementation supports the `tree-sitter` query language, `comby`-like syntax, as well as regular expressions.
2. We enable match-replace languages to be interleaved in the graph (e.g., the first rule uses regular expressions, while the second uses `comby`). Some tasks might be accomplished using simple regular expressions, but for other `comby` might be a better option.
3. We support composition of match-replace rules using a set of language-agnostic filter primitives. The goal is to enhance rule precision by leveraging the surrounding code context. Instead of writing a match-replace rule, the idea is to write multiple rules and only transform the code if all conditions are satisfied. In this sense, filters act as a set of pre-conditions.

#### 5.2.2.A DSL Motivating Example

Figure 5.2 illustrates a program to automate the migration process outlined in Figure 5.1 in our proposed language. In this example, rules are expressed in the `comby` syntax as explained in Chapter 4. A program in our DSL is a graph of match-replace rules. The graph has a source/seed rule `Replace Import`, i.e., this rule initiates and triggers the code transformations by replacing the `log4j` import. `Replace Import` is connected to another rule `Update Logger Creation` with an edge labelled `File`. This means that the rule `Update Logger Creation`'s application is restricted to the files affected by the `Replace Import` rule.

Furthermore, `Update Logger Creation` has two outgoing edges: (1) `Update Info` - for updating the info API usage between the libraries, and (2) `Append Import if Needed` - for appending the import of LoggerFactory if necessary. First, we seek to update the info API usage only for the `Logger` field that was previously updated. Thus, `Update Info` takes as input the `:[obj]` name from the previous rule, as indicated by the `using` keyword. The value of `:[obj]` will be instantiated at runtime based on the previous rule. Moreover, the rule will only be applied within the same `Class` as indicated by the edge label. Second, the import

```
<rule_graph> ::= <rule>+ <edge>*
<edge> ::= from string to (string, scope)
<scope> ::= Global | File | n-Ancestors
        | Method | Class
<rule> ::= name string match match
     [replace [template_variable] with <replace>]
     [where <filters>]
     [using <holes>]
     [is_seed bool]

<replace> ::= string <replace>
          | template_variable <replace> | <>
<filters> ::= enclosing match [<contains>] <filters>
          | not_enclosing match <filters> | <>
<contains> ::= contains match  [at_least int] [at_most int]
          | not_contains match | <>
<holes> ::= template_variable <holes> | <>
```

**Figure 5.3:** Syntax of our DSL for cascading code transformations. The elements inside square brackets are optional. The symbol `match` is an expression for pattern matching (e.g., `comby`); the symbol `template_variable` represents named capture groups from the `match` pattern.

is appended to the `File` where the `getLogger` API was applied as indicated by the edge between `Update Logger Creation` and `Append Import if Needed`.

Notice that the rule `Append Import if Needed` uses an extra clause called *enclosing*. Enclosing is a filter that ensures the import is not added to the file twice. In this case, the filter is written in a combination of `tree-sitter` and regular expressions. The *enclosing* pattern `(program @p)` is a *tree-sitter* query that matches the entire source file. The *not contains* clause specifies a regular expression to check against enclosing source file. The rule will only match if the source file does not already contain the import statement.

## 5.3 Language Syntax and Overview

Figure 5.3 describes the grammar of our DSL. A program in the DSL is a graph of match-replace rules. The rule graph is captured as a list of *directed* and *labelled* edges. Each node represents an individual transformation rule that structurally matches and rewrites code. Rules can also just match code without transforming it. The edges between rules specify *which* rule to apply next and the scope where it should be applied.

### 5.3.1 Edges

As shown in Figure 5.3, the edges are *directed and labelled*. Each *edge* connects either two rules or a rule to a rule group, defining the order in which they should be applied, akin to the andThen operator[1]. The edge label specifies the *scope* of application, selecting the portion of the code base upon which the target rewrite rule is applied, with respect to the code that the source rule matched. For example, given an edge from $\mathcal{R}_1 \xrightarrow{\text{method}} \mathcal{R}_2$, reads as "apply $\mathcal{R}_1$ and then apply $\mathcal{R}_2$ within the enclosing *method* where $\mathcal{R}_1$ was applied".

The DSL supports two *language-agnostic* predefined scopes as shown in Figure 5.3. 1. `Global` the target rule is applied across the codebase, 2. `File` the target rule is applied in the enclosing file. It is also possible to support other *language-specific* scopes that depend on the granularity of the internal representation of code

[1] https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#andThen-java.util.function.Function

within the implementation. The current implementation represents code internally with `tree-sitter` [93], and supports three other scopes: 3. *`n-Ancestors`* the target rule is applied to the parent parse tree nodes of the code fragment that the origin rule matched, 4. *`Method`* the rule is applied to the enclosing method where the preceding rule was applied, and 5. *`Class`* scope refers to the enclosing class. Note that the language specific scopes are set up only once per language.

### 5.3.2 Rules

Besides the name, a match-replace rule has 4 major components 1. *match* - a pattern to match source code, 2. *replace* - a pattern to rewrite the matched code, 3. *filter* - to filter out certain matches based on the surrounding code, and 4. *holes* - variables referenced in the rule, these are filled at run time and serve as the *dynamic* component of the rule. Furthermore, a rule could be a `seed_rule`. These seed rules are entry points to the graph. This graph is traversed in a depth-first manner at each location where the rule was applied. A valid rule graph contains at least one *seed rule*.

#### 5.3.2.A Match.

The *match* expression is a declarative pattern that captures a code snippet with a specific *structure* or *shape* (based on its parse tree). The match also labels portions of the matched parse tree like the *named captured groups* [161] in regular expressions. Our DSL can support multiple structural matching languages, as long as they support named capture groups (used to label portions of the code). The current implementation supports concrete patterns, structural queries, and regular expressions.

*Concrete Patterns:* A concrete pattern is a string with template variables / holes, that is matched to concrete syntax nodes [162] from the program's parse tree [2]. Formally, let *s* be a concrete pattern containing holes of the form `:[var1]`, where each hole can represent syntactically valid sub-trees. A Concrete Syntax Tree (CST) node *t* matches *s* if, traversing *t* in depth-first order yields leaf nodes with a string representation that aligns with *s* from left to right. Each hole can represent entire sub-tree structures (i.e., multiple sequential leaf nodes under an internal node). This paradigm of matching is supported by multiple other tools (e.g., [23, 75]). PolyglotPiranha adopts the syntax proposed by [23] in their tool `Comby`. However, our concrete patterns have stricter semantics compared to `Comby`. In our concrete pattern, a template hole, `:[x]`, matches whole syntactic structures / CST nodes, whereas `Comby` templates can represent arbitrary strings. Figure 5.4 shows two examples.

*Structured Query Language:* A query consists of one or more patterns, where each pattern is an *s-expression* that matches a certain set of nodes in a parse tree. These queries capture the structure of the target pattern in terms of AST node types and string based predicates. This paradigm is programming language agnostic,

[2] We use Concrete Syntax Trees (CST) over Abstract Syntax Trees (AST) because we must preserve all syntactic structures within the source code, which are necessary for source code matching

| Rule | Matched code snippet | Capture Groups |
|---|---|---|
| `match :[x].info:[args]` | `logger.info(example)` | `x: logger, args: example` |
| `match import org.apache.:[name]` | `import org.org.apache.log4j.Logger` | `name: log4j.Logger` |

**Figure 5.4:** Example rules using concrete patterns applied to motivation example in Figure 5.1.

| Rule | Matched code snippet | Capture Groups |
|---|---|---|
| `match (method_invocation object: (identifier) @obj name: (_) @m arguments: (object_creation))` | `logger.info(new StringBuilder("append0"))` | `obj: logger, m: info` |
| `match ((field_declaration (name: (identifier) @obj (value: (method_invocation name: (_) @m))) (#eq? @m "getLogger"))` | `Logger logger = Logger.getLogger(A.class)` | `obj: logger, m: getLogger` |

**Figure 5.5:** Examples of rules using simplified structural queries applied to motivation example in Figure 5.1.

and is supported by systems like `tree-sitter`. PolyglotPiranha supports the s-expression based tree-sitter queries [163]. Figure 5.5 shows two examples.

Each matching paradigm has distinct advantages and disadvantages. By construction structural queries are more precise than concrete syntax because they can leverage node-types or absence of particular nodes, and therefore leave less room for ambiguity (e.g., it is possible to differentiate between a field and a local variable declaration). For example, matching method declarations is easier with structural query, because we would not need to account for all its syntactic variations (e.g., modifiers like `public, static, final`) like in concrete syntax. In contrast, matching API invocation pattern like `logger.info(example)` (from Figure 5.1) the *concrete pattern* is convenient and more succinct. The structural query for this pattern is verbose, and requires knowledge of the target language's grammar. Regex matching is more suitable for semi-structured documents like markdown files. Note that PolyglotPiranha is not tied to these three languages, more can be supported.

#### 5.3.2.B Replacement.

The replacement pattern decides on how a matched code snippet should be transformed. It is possible to either replace the entire matched code or just segments identified by a named capture group. The replacement expression / pattern can be seen as partial function that is instantiated at run time by substituting a referenced named groups or template variables with their values from either the initial match in the rule, or inputs to the rules declared with the `using` keyword (i.e., code snippets captured in previous rule applications). In Figure 5.6, we show two examples of replacement rules. In the first one, the code is only partially rewritten. In the second example, the matched code is completely deleted because no target node is specified.

| Rule | Source Code Update |
|---|---|
| `match :[object].info(:[args])`<br>`replace :[object] with other_:[name]` | `logger.info(new Object())`<br>`other_logger.info(new Object())` |
| `match import org.slf4j.:[rest]`<br>`replace with -` | `- import org.slf4j.Logger`<br>`- import org.slf4j.LoggerFactory` |

**Figure 5.6:** Examples of replacement rules using concrete syntax. Notice that in the first example, the code is only partially rewritten. Whereas in the second example, the entire code snippet is deleted.

| Rule | Source Code Update |
|---|---|
| `Delete unused local variable`<br>`match :[var_name] = :[rhs];`<br>`replace with -`<br>`where enclosing (method_declaration)`<br>`contains :[var_name] atmost 1` | `def some_python_function() {`<br>`- int unused_variable = 42`<br>`  execute()`<br>`}` |
| `Add import statement if absent`<br>`match (import_declaration) @p`<br>`replace with @p \n import org.slf4j.LoggerFactory;`<br>`where enclosing_node (compilation_unit)`<br>`not_contains import org.slf4j.LoggerFactory` | `import org.slf4j.Logger;`<br>`import org.slf4j.LoggerFactory;`<br><br>`class A {`<br>`  Logger logger = LoggerFactory.getLogger(A.class);`<br>`}` |

**Figure 5.7:** Example rules using *filters*. Note how these rules leverage both *concrete pattern* and *structural query*. In the first example, we use a contains filter inside the enclosing method declaration. This allows us to check if a variable is used only *once*. If this is true, the usage corresponds to its declaration, and thus, can be safely deleted. In the second example, the import is added only if it is not already in the code, as indicated in the not contains predicate. Since the import is already present, the code is not rewritten.

#### 5.3.2.C Filters.

To make the *rules* more precise and context-aware, our DSL provides `filters` to control the application of a rule based on the surrounding code. First, the candidate code to transform is checked against the `matcher` of the rule. Then, at each matched location, the filters will check if the surrounding code of this location satisfies certain criteria.

There are two primitive filters: 1. `enclosing` – checks if the primary match is enclosed by a parse tree node that satisfies the given matcher, and 2. `not_enclosing` – checks if the primary match is *not* enclosed by parse tree node that satisfies the given matcher. The `enclosing` filters can be further refined by specifying `contains` and `not_contains` expressions. The `contains` (`not_contains`) expressions specify `matchers` that should (not) match at least once inside the `enclosing_node`. The user can also specify the frequency of these matches with `at_least` and `at_most` attributes.

**Holes.** These serve as dynamic components within a rule. They describe input variables to the rule. At run time, their corresponding values are populated from a symbol table (which maintains the bindings from named captured groups to code snippets from current and previous applications). The example in Figure 5.2 showcases the usage of these `holes`. The rule `Update Info` declares the hole `:[obj]`, which will be instanti-

ated during the transfromation based on the code matched in the previous rule `Update_Logger_Creation`.

## 5.4 Language Runtime

### 5.4.1 Algorithmic Overview

Algorithm 3 provides a high level overview for the language implementation and runtime. The core idea is to maintain a queue of *seed rules*, and traverse the graph and the files in the codebase starting from each seed rule. First, we validate the rule graph to prevent unexpected behavior using a data-flow analysis and syntactic checks on the rules (Line 1). After the validation, we push the seed rules into a global queue and initialize an environment / symbol table with the input substitutions (Line 4 - 5). The environment is used to store both the initial set of substitutions as well as the captured groups of from rule executions, which can be used as dynamic elements in subsequent rules. Each seed rule is applied across the entire codebase recursively in a depth-first fashion (Line 6), until no rules match (Line 8 - 15). For each *relevant* file (e.g., a file that is likely to contain the match template of the rule, see Section 5.4.1.C), we invoke `ExecuteRuleGraph` (Algorithm 4). In this step, the tool traverses over the CSTs and transforms the source code. For each match, it explores the rule graph and stacks the rules in a DFS-manner (Line 12), applying them exhaustively within the scope. The function `ExecuteRuleGraph` is *not pure*, it updates the environment, transforms the source code in-place, and pushes new rules into the queue ($Q$). We detail each function of the algorithm more thoroughly in subsequent sections.

#### 5.4.1.A Graph Validation

The first step in the core algorithm is to verify the graph (Line 1). In our implementation, PolyglotPiranha statically validates the constructed graph to prevent unexpected behavior when the graph is applied to the codebase. First, PolyglotPiranha checks if the individual rules' matchers and filters are well-formed. For example, PolyglotPiranha ensures that each regex compiles and that each s-expression parses correctly according to the language's grammar. It also conducts a data-flow analysis to ensure that no path in the graph traversal leads to a rule where an input variable is not initialized correctly. This is implemented as a definite assignment analysis [164]. If the graph is incorrect, PolyglotPiranha alerts the user to prevent panics that could result from accessing undefined variables.

#### 5.4.1.B Environment

The environment is a simple symbol table, which is initialized with the substitutions from the program (Figure 5.3). Rules can access symbol table variables if they have been declared. If a rule is triggered and a match is found, the symbol table is updated by binding the matched source code to the corresponding named cap-

**Algorithm 3** Core procedure for transforming code given a graph of rules

**Input:**
($\mathcal{R}$ : RuleGraph, $\mathcal{S}$ : substitutions)
$\mathcal{C}$ : path to codebase

```
1: if ¬VALIDATE(R, S) then
2:     return
3: end if
4: Q ← SEEDRULES(R)
5: env ← S
6: while NOTEMPTY(Q) do
7:     rule, _ ← POP(Q)
8:     loop
9:         isApplied ← false
10:        for file in RELEVANT(C, rule, env) do
11:            isApplied ∨ =
12:    EXECUTERULEGRAPH(rule, file, R, Q, env)
13:        end for
14:        if ¬isApplied then
15:            break
16:        end if
17:    end loop
18: end while
```

**Algorithm 4** EXECUTERULEGRAPH function

```
1: function EXECUTERULEGRAPH(rule, file, R, mut Q, mut env)
2:     rulesStack ← [(rule, file)]
3:     isApplied ← false
4:     while NOTEMPTY(rulesStack) do
5:         rule, scope ← POP(rulesStack)
6:         rule ← INSTANTIATE(rule, env)
7:         while HASMATCH(rule, scope) do
8:             match ← GETMATCH(rule, scope)
9:             isApplied ← true
10:            APPLYEDITS(match, rule, mut env)
11:            for rule, scScope in SUCCESSORS(rule, R) do
12:                if scScope ≡ GLOBAL then
13:                    PUSH(Q, (rule, GLOBAL))
14:                else
15:                    scope ← RESOLVE(match, file, scScope)
16:                    PUSH(rulesStack, (rule, scope))
17:                end if
18:            end for
19:        end while
20:    end while
21:    return isApplied
22: end function
```

tured group in the symbol table. If a variable already exists in the symbol table, its entry gets over written. Therefore a rule always gets instantiated with the most recent binding of the referenced symbol from the environment. This kind of dynamic variable scoping can also be observed in languages like `LaTeX` or `Bash`.

#### 5.4.1.C Relevancy check for performance

In rewriting large code bases, repeatedly parsing the entire codebase is inefficient, especially in monorepos with millions of lines. The goal of the function `relevant` is to optimize code rewriting by only parsing files whose content matches the concrete values assigned to the holes of the *global rules* (Line 10). In practice, the concrete values to the input substitutions are used to filter out files that are not relevant to the transformation using string matching. This simple insight improves POLYGLOTPIRANHA's overall performance. The implementation of POLYGLOTPIRANHA further boosts this by parallelizing the lookup using fork-join frameworks (like `Comby`). Note that, POLYGLOTPIRANHA circumvents this optimization for holes that are referenced inside the `not_contains` or `not_enclosing` clause.

### 5.4.2 Rule Graph Execution

Algorithm 4 describes the procedure EXECUTERULEGRAPH that applies a given *rule* across a *file*. Each time a *seed rule* is triggered, we initialize a *stack* (`ruleStack`) for depth-first traversal of the rule graph (Line 2). Then, we pop rules from stack and apply each rule exhaustively within the specified scope (until `hasMatch` is `false` as

shown in Line 7). For each match, we update the environment with the new capture groups and transform the source code by applying the rule (Line 10). Finally, we add the successors of the current rule in the graph to the local stack or the global queue, and continue this until fix point (Lines 11 - 18).

#### 5.4.2.A Optimizations.

PolyglotPiranha uses the `tree-sitter` [93] framework for parsing the source code. PolyglotPiranha maintains only one parse tree *object* in its memory, and updates this object sequentially leveraging the `tree-sitter`'s incremental parsing feature. This eliminates the need to parse the file again from scratch after the rewrite, thus optimizing PolyglotPiranha's overall performance. Additionally, to minimize the impact on the parse tree, by default our approach 1. orders the rules from inner to outer scope: starting from the parent, to method, class, file, and finally to global scope, and 2. rewrites code bottom up. In the future, we plan to support alternative transformation strategies.

## 5.5 Evaluation

Our language implementation is merged into the PolyglotPiranha repository, which is maintained on GitHub. The tool is used internally at `Uber` with multiple use cases. In our evaluation, we aim to show key desirable properties of the language for our use case. To do this, we evaluate PolyglotPiranha on three case studies related to migration and code cleanup. In particular, we answer the following research questions:

**RQ1.** **[Expressiveness]** How expressive is the DSL for real-world code transformation tasks? *We assess this through three case studies. We highlight the complexity of each, and how to encode it in the DSL.*

**RQ2.** **[Effectiveness]** How effective is our language at automating code changes? To what extent is it useful in practice? *We run the above tools across `Uber`'s proprietary codebase, and measure the percentage of Pull Requests (PRs) that pass Continuous Integration (CI) and are merged without manual intervention. For PRs with intervention, we measure the `LoC` changed by tools versus developer.*

**RQ3.** **[Comparison with state-of-the-art]** How do tools built upon the DSL compare to similar tools built upon state-of-the-art frameworks? *We compare the PolyglotPiranha-based implementation against the imperative variants developed upon `ErrorProne` [91] and `OpenRewrite` [92], and against its declarative variants developed upon `Comby` [23] (a lightweight tool). We compare implementations in terms of size, complexity and performance.*

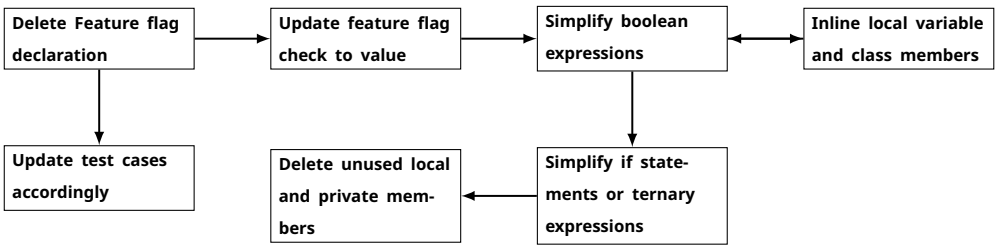



**Figure 5.8:** Strongly connected components of the rule graphs for feature flag cleanup. The graph structure is language-agnostic. Implementations accross languages require some adapations.

### 5.5.1 RQ1. Expressiveness

#### 5.5.1.A Experimental Setup

To showcase the *expressiveness* of the DSL, we present *three* real-world case studies where we automate complex code transformation tasks using PolyglotPiranha. In each case study, we highlight the complexity of the task, and how the DSL can be used to encode it. We chose these three case studies because they are high-impact tasks crucial to `Uber`'s operational needs and they are representative of the tasks that `Uber` or other software companies would want to automate. Moreover, these tasks are not trivial to automate using existing frameworks.

#### 5.5.1.B Case study: Stale Feature Flag Cleanup

Feature flagging is a widely adopted and highly encouraged practice at `Uber` [3], and other major software companies [165, 166]. It allows developers to modify configurations without redeploying, supporting A/B testing in production. However, feature flags often become stale, and retaining them beyond their original purpose can lead to technical debt. Therefore, it is important to automate their removal. Indeed, researchers [90] have developed the `Piranha` tool for this purpose. `Piranha` is built on top of the `ErrorProne` [91] frameworks for `java` and `SwiftSyntax` [167] for `Swift`. However, `Uber`'s codebase uses `Kotlin` and `Go` too. Instead of developing two new language-specific tools, we used PolyglotPiranha to implement this transformation as ***one*** tool supporting `java`, `Kotlin`, `Swift` and `Go`.

Figure 5.8 shows the strategy that we implemented for automating the cleanup of stale feature flags at `Uber`. Each node in this figure is a strongly connected component or sub-graph of the original large graph implementing the transformation. Here, each subgraph is a cleanup category. For instance, `Simplify boolean expressions` contains rules that simplify nested boolean expressions with conjunctions, disjunctions and negations. These rules are recursively applied until the expression cannot be further simplified. It should be noted how the `Simplify boolean expressions` and `Inline local variables and members` call each other, until no more simplification is possible. The `Cleanup tests` sub-graph is particularly interesting. In this sub-

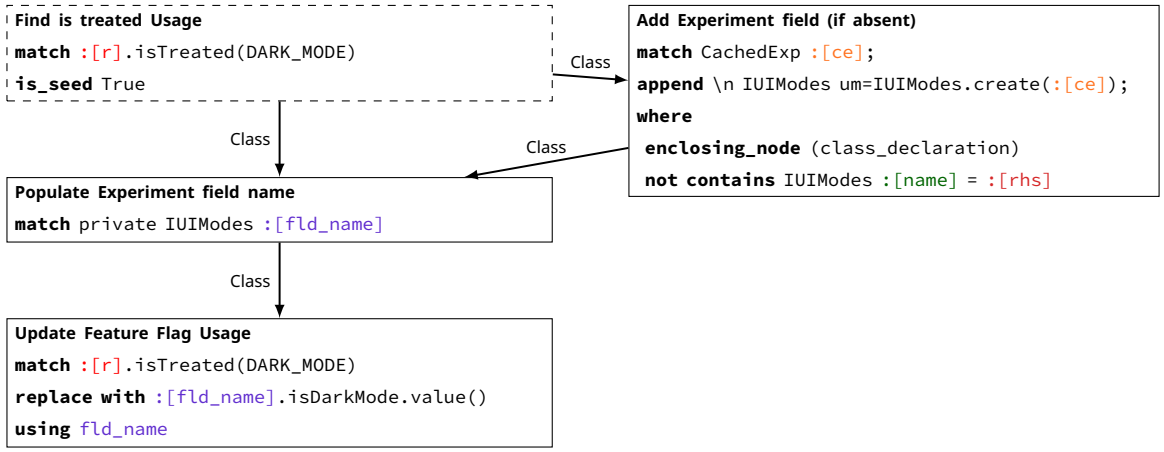
[3] In fact, our motivating example is a simplified version of feature flag cleanup we performed internally.

```
1  - public enum IUIModesEnum {
2  + public interface IUIModes {
3      - DARK_MODE,
4      + @Param(key="DARK_MODE")
5      + BoolParam isDarkMode();
6      - LIGHT_MODE,
7      + @Param(key="LIGHT_MODE")
8      + BoolParam isLightMode(); }
```

**(a)** Example migration from enum-based feature flag declaration to annotations.

```
1  class Consumer {
2    CachedExp ce = new Experiment();
3  + IUIModes um = IUIModes.create(ce);
4    public String color() {
5  - return ce.isTreated(DARK_MODE)
6  + return um.isDarkMode().value()
7      ? "Black"  : "White";
8  }
9  }
```

**(b)** Source code update after the migration of enums to interfaces as shown in Figure 5.9a.




**(c)** Part of the original rule graph that migrates usages of the `isTreated` API. The input substitutions in the bottom right instantiates this graph to migrate the DARK_MODE feature flag described in this figure.

**Figure 5.9:** Experimentation API usage update after the migration from `enum`-based feature flag declarations.

graph we identify all the tests that explicitly set the feature flag to a specific Boolean value. If the set value is the same as the status of the feature flag we elide the setter, else we delete the test case.

#### 5.5.1.C Case Study: Experimentation API Migration

The *Experimentation* team at `Uber` developed a new *feature flagging* API to support its growing needs. It was imperative for `Uber` to transform thousands of lines of their `Android` code to use this new API.

Figure 5.9 showcases the code changes required for the migration. The previous *feature-flag* API declared feature flags using `enum` data types. To adapt the code to the new API, these enums need to be rewritten as *annotated abstract methods* (as shown in Figure 5.9a). These annotations were added to specify metadata information for a feature flag such as *key* and *namespace*. After migrating the `enum` to an interface, this change has to be propagated. For example, consider the feature flag usage in Figure 5.9b. Previously, the `isTreated` method (Line 5) was invoked to check the status of the feature flag by passing the enum `DARK_MODE`, declared in Figure 5.9a. However, with the new design clients are expected invoke the feature flag method `isDarkMode()` as shown in Line 6.

In practice, this migration has to accommodate many other *caveats*. To complete this migration, it is

```
company_kotlin_android_module(
  name = "src_release",
  plugins = [
- //libraries/compiler:processor",
    "//libraries/utilitites",
  ],
+ kotlinc_plugins = [
+ "//libraries/processor-kt:processor"],
  tests = [":test_release"],
  visibility = ["PUBLIC"],
)
```

**(a)** Changes to the BUCK file. Here the java dependency is replaced with the Kotlin counterpart

```
- import com.co.ParameterUtils
  interface UIParams{

  @JvmStatic
  fun create(cp:CachedParams): UIParams =
- ParameterUtils.create(UIParameters::class.java,cp)
+ UIParamsProvider.create(cp)
   }
  }
```

**(b)** Changes in the source code illustrating the usage of the new Kotlin-based processor

**Figure 5.10:** Examples of modifications in the BUCK and Kotlin files for the annotation processor migration.

necessary to also add new fields (e.g., `IUIModes` (Line 3, Figure 5.9b). This is handled by writing two rules as shown in Figure 5.9c : 1. `Add Experiment field` - adds a field of type `IUIMode` (if absent), and 2. `Populate Experiment field name` captures the name of the field of type `IUIMode`. The field name (i.e.`:[fld_name]`) is used in the following rule `Update Feature Flag Usage`, which is the actual rule used to replace the `isTreated` API. Other nuances include deleting consequently unused members and imports and adapting test cases accordingly.

#### 5.5.1.D Case Study: Annotation Processor Migration

The goal of this migration is to transition the `Android` codebase from a `java`-based annotation processor to a `Kotlin`-based system to improve overall performance. The changes required for this migration are described in Figure 5.10. This migration requires changing all the build configurations (written in BUCK [168]) to be adapted by replacing the old processor dependency with the new one as depicted in Figure 5.10a. Besides the build files, it is necessary to migrate all `Kotlin` files that initially used the `java` processor (shown in Figure 5.10b). For example, Figure 5.10b shows how `ParameterUtils.create` is replaced with a `Kotlin` equivalent method, `create`, and the unnecessary import statement is deleted.

### 5.5.2 RQ2. Effectiveness and Usefulness

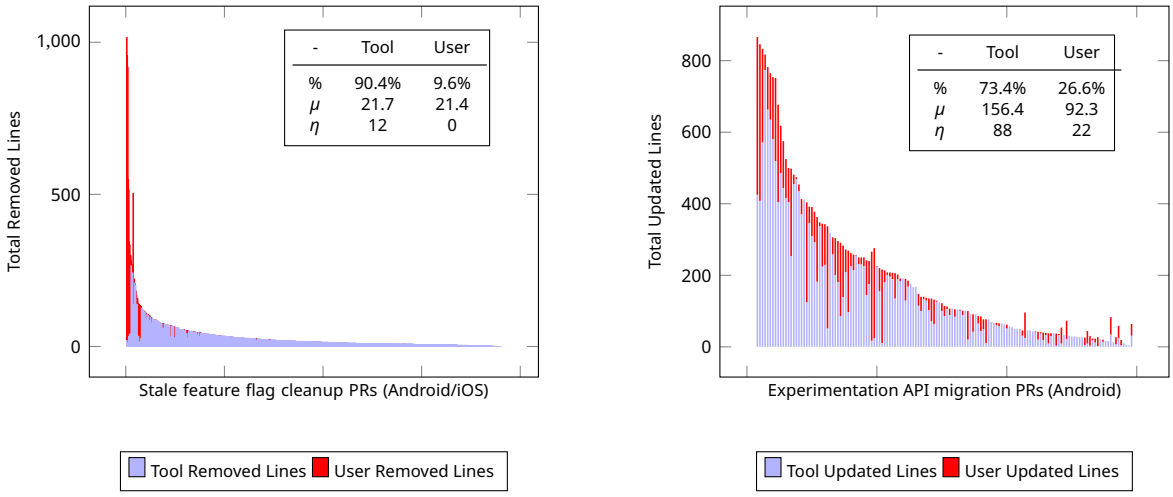
#### 5.5.2.A Experimental Setup

To evaluate the effectiveness of PolyglotPiranha's framework, we show its effectivness on the three case studies above by applying them to `Uber`'s proprietary code-bases. Specifically, the `Android` codebase is composed of 7.5M `LoC` of `java` and 2.5M `LoC` of `Kotlin`, while the `iOS` codebase is composed of 7.5M `LoC` of Swift. The PRs produced by our tools are reviewed by the appropriate teams, and merged if they pass the *Continuous Integration* checks and tests. The PRs that fail CI are expected to be manually fixed by the respective team before merging. Note that this data represents months of company wide-effort.

| Application | Language | Effectiveness | | Usefulness | | |
|---|---|---|---|---|---|---|
| | | # PRs | # PRs (CI passes) | # PRs (Accepted) | # files updated | #+/- Lines |
| Stale Feature Flag Cleanup | Java & Kotlin | 2515 | 1413 | 817 | 2952 | +15032 /-107635 † |
| | Swift | 2186 | 1309 | 614 | 1733 | + 21230 /-104721 ‡ |
| Experimentation API migration | Java | 155 | 89 | 155 | 2146 | +19157 /-19041 § |
| Annotation processor migration | Kotlin & Python | 25 | 25 | 25 | 2042 | +2809 /-3897 ‖ |

† 85.7% was automated ‡ 95.3% was automated § 73.4% was automated ‖ 100% was automated

**Table 5.1:** PRS created and merged by the tool, as well as the % of LOC automatically deleted for each.




**(a)** Lines deleted by feature flag cleanup PRs

**(b)** Lines updated by API migration PRs

**Figure 5.11:** Lines deleted/updated by tool (blue) vs users (red)

#### 5.5.2.B Results

Table 5.1 summarizes the overall results we obtained by running PolyglotPiranha based tools over our proprietary corpora. For each application, it reports the number of PRs created, PRs accepted (and merged), and PRs that pass the Continuous Integration checks and tests. At large, the three tools produced 4881 PRs in the last six months of which 1611 have been accepted and merged into the main codebase at the time of writing this paper. Particularly, for *stale feature flag cleanup* our acceptance rate is 52.5% (of PRs that pass CI) while for the migrations it is unsurprisingly 100% (because the migrations were orchestrated centrally). These PRs have deleted over 200k LoC of dead code and migrated over 20k LoC of old code to use the new APIs.

**A – Stale feature flag cleanup** The data for this experiment was collected between April and November of 2023. PolyglotPiranha created a total of 4701 PRs, and reviewers did some kind of activity on 1727 (36.7%) of the total number of PRs. These activities include, accepting the PR and merging it, commenting the PR, or patching the PR before accepting it. There are still 1410 PRs that pass all CI checks and are still in queue for

review. Further, the reviewers have marked 114 PRs as `Needs Changes` status indicating that the they expect extra cleanup from the tooling. For most of these PRs, the reviewers have reported issues with new features and bugs. The reviewers abandoned 182 PRs, to assert that the cleaned up feature flags are not stale.

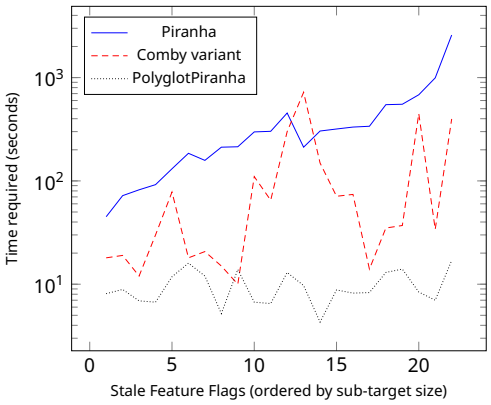
We observed that 56.2% of all the `Android` PRs and and 59.9% of the `iOS` PRs passed all CI checks. `Uber`'s CI not only builds and tests the change, but it also employs over a hundred linters and bug-checkers to ensure the quality of the change meets the `Uber`'s high standards. These checkers ensure there are no unreachable and unreferenced elements (e.g. `UnusedMethod` check [169]), no sub-optimal code (e.g. `ComplexBooleanConstant` check [170]) and no nullability errors [171].

The tool deleted 85.7% and 95.3% of all the total deleted lines across the `Android` and `iOS` codebases (90.4% gross) respectively across all merged PRs, as shown in Figure 5.11a. We observed that 75.9% of the PRs that were merged required no user intervention. However when the developer did intervene, they deleted a lot of code before merging the PR, hence the mean number of lines deleted by user is skewed ($\mu = 21.4$, $\eta = 0$). In few outlier cases developer deleted more than 900 lines of code. Probing further into these outlier PRs, we discovered that developers had removed a collection of top-level classes that were guarded by the flag. Some of these scenarios will be incorporated into the next version of our tool. However, very precise and general support for such cleanups is impractical in our lightweight approach.

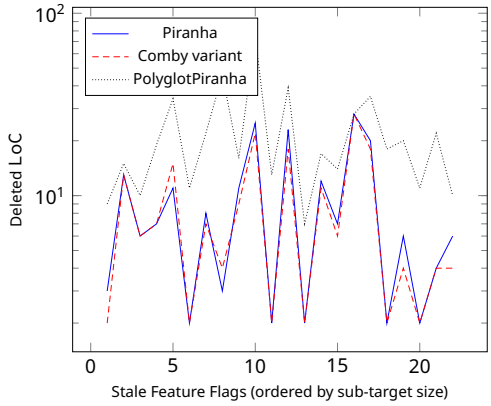
**B – Experimentation API** For the *Experimentation API migration*, we observed that 89 (59.9%) PRs passed all CI checks. The main reason migration PRs to fail was non-standardized usage of the API and usage of some specific API patterns that were not automated. Nonetheless, the tool still automated 73.4% of all lines deleted. The migration was driven centrally by the team, therefore the all the PRs were immediately acted upon after creation. The team reviewed these PRs, patched them if necessary and merged them.

The tool migrated 73.4% of the total lines deleted, however we observed that more than 74.8% PRs needed some manual intervention. In these cases developers on average updated another 92 lines upon the changes proposed by the tool. We observed that `Uber` developers also made manual changes to the PRs that pass CI. These changes include class deletions, removing unused data files, updating comments and method names. While refining rules can resolve certain scenarios, some require symbol or type information, and others, such as method renaming and updating documentation, are beyond the scope of traditional tools. We also observed that the team knowingly used the tool to perform partial migrations even for cases where all APIs were not supported. The small spikes towards the tail end of the chart show these scenarios.

**C – Anntoation processor migration** All the 25 PRs for this migration passed CI and were merged automatically, without any user intervention.




**(a)** Time required to perform the cleanup. Flags are ordered by target size.




**(b)** Lines of code deleted for each tool. Flags are ordered by target size.

**Figure 5.12:** Comparative analysis of `Comby`, `Piranha`, and `PolyglotPiranha` for stale feature flag clean up.

### 5.5.3 RQ3. Comparison with state-of-the-art code rewrite frameworks

#### 5.5.3.A Performance

**A – Experimental Setup** We compare PolyglotPiranha-based stale feature flag cleanup against `Piranha` [90] and an equivalent we develop based upon `Comby` [23]. The `Comby` implementation has 29 rewrite rules for `java`. It was particularly easy to develop the `Comby` variant because PolyglotPiranha's concrete syntax DSL is inspired by `Comby`. For this evaluation, we chose 24 stale feature flags randomly from the PRs that 1. passed CI but were not accepted (at the time of writing this paper) 2. were used in `java` files (because `Piranha` only supports `java`). Note that we only chose 24 feature flags because it takes significant manual effort to integrate `Piranha` within our infrastructure due to `Piranha` depending on compilation[4]. For each feature flag, we noted the affected sub-targets and their sizes. We then applied the three tools across the sub-targets and the execution time was recorded. These experiments were performed on an enterprise-class VM in Google Cloud Platform. Note that we neither compare the quality of the cleanups nor precision because by construction `Comby` uses a more loose representation of code, based on Dyck-extended grammars [23] (i.e., balanced parenthesis grammars), whereas PolyglotPiranha uses language-specific grammars from the `tree-sitter` reportoire, hence PolyglotPiranha transformations are more powerful and precise. Conversely, `ErrorProne` and `OpenRewrite` can leverage semantic information like symbol/name resolution, for higher precision and applicability, but are not polyglot.

**B – Results** The line chart in Figure 5.12a shows the performance of each of the tools for the set of flags we identified above (ordered by the size of the corresponding *sub-targets*, ranging from 1.2K to 1.8M LoC). PolyglotPiranha took an average of 9.74 ± 3.46 seconds, `Comby` 121.67 ± 179.03 seconds (12.32×), and `Piranha` 413.91 ± 521.94 seconds (42.5×). We can see `Piranha`'s execution increases almost linearly with the target

[4]While Piranha was developed and was previously integrated at Uber, however, both Uber's feature flag API and developer infrastructure have changed since then

**Table 5.2:** Comparison of PolyglotPiranha against existing tools

| | | Bug Fixes | | | Feature Flag Cleanup | |
|---|---|---|---|---|---|---|
| **Tool** | **Metric** | CWE-338 | slf4J | java.security | Android | iOS |
| PolyglotPiranha | LoC | **68** | **31** | **23** | **654** | **1156** |
| | # rules | 4 | 3 | 3 | 31 | 42 |
| Error-Prone | LoC | - | - | - | 3467 | - |
| SwiftSyntax | LoC | - | - | - | - | 1316 |
| OpenRewrite | LoC | 145 | 87 | 92 | - | - |
| Comby | # rules | - | - | - | 29† | - |

† This feature flag cleanup variant was developed for the experiments.

size (due to the fact that `Piranha` relies on building the target). PolyglotPiranha and `Comby` depended on the number of passes and files affected for the refactoring. The fact that PolyglotPiranha is faster than the `Comby`-based variant is surprising because `Comby` has a string based matching approach with minimal overhead. These results can be attributed to the fact that `Comby` has no sense of ordering between rules (nor scope), therefore, the match-replace rules are applied across the entire subtarget. PolyglotPiranha's performance is also attributed to optimizations discussed in Section 5.4.1.C and 5.4.2.A.

Figure 5.12b shows the number of lines deleted by each of the tools for the same set of flags (in the same order). PolyglotPiranha deletes more lines of code because it's able to delete trailing commas and comments. Note that we manually vetted that there are no over deletions in these PRs. In summary, PolyglotPiranha is consistently faster while deleting more lines than its imperative alternative `Piranha` and a lightweight `Comby`-based alternative.

#### 5.5.3.B Expressiveness and Ease-of-use

**A – Experimental Setup** We compare the implementation of PolyglotPiranha-based tools against their imperative variants. Specifically, we compare the PolyglotPiranha-based Stale Feature flag cleanup program against the implementation of `Piranha` [90]. Further we also encode three *pre-existing* code transformation recipes developed by *professional tool builders*, specifically `OpenRewrite` - 1. (`JHipster Upgrade`) Fix CWE-338 with SecureRandom [172] 2. (`Slf4j`) Loggers should be named for their enclosing classes [173] 3. (`java.security`) Use secure temporary file creation [174]. The selected patterns are 1. related to a popular `java` library 2. involve multiple interdependent changes 3. have associated test cases for validation 4. clearly fix a bug or security vulnerability.

**B – Stale feature flag cleanup.** As discussed in Section 5.5.3.A, `Piranha` is a stale feature flag cleanup tool with multiple implementations, one for each language supported. This is because `Piranha` is built upon language-specific imperative frameworks for code analysis and rewriting. To compare the expressiveness and conciseness of both approaches, we qualitatively and quantitatively compare the `Piranhajava` and `PiranhaSwift` variants against their PolyglotPiranha-based counterparts.

`PiranhaJava` is built upon the `ErrorProne` [91] framework, whereas `PiranhaSwift` uses `SwiftSyntax` [167]. Table 5.2 (right) shows that PolyglotPiranha based approach is significantly more concise in terms of `LoC`. Moreover, rules can be re-used across languages. PolyglotPiranha-based `Swift` variant is more powerful than `PiranhaSwift` (e.g., supports variable inlining and cleanup of the unused members).

In contrast, our `Comby` implementation for feature flag cleanup in Java comprises 29 rules. Due to `Comby`'s limitations, we were unable to express 10 transformations from our PolyglotPiranha implementation, including inlining singly-used boolean variables, deleting unused fields and variables, removing unnecessarly nested blocks, and deleting files under certain conditions and enum blocks. Despite this, the rule count difference is minor: 31 for PolyglotPiranha versus 29 for `Comby`. This is because PolyglotPiranha allows for the use of different, more powerful transformation languages. For instance, `tree-sitter` queries provide a syntax for complex alternations. Therefore, the `Comby` variant ends up being more verbose, requiring additional rules for the same task.

**C – OpenRewrite** `OpenRewrite` project is a semantic code search and transformation ecosystem. Its platform allows writing code transformation recipes for common framework migration and stylistic consistency tasks. We picked three relevant recipes written by professional developers, corresponding to high-impact transformations. We implemented the same refactoring actions using PolyglotPiranha. Table 5.2 shows the `LoC` count and number of rules for both PolyglotPiranha and `OpenRewrite` recipes. Our implementations pass the tests of the `OpenRewrite` recipes.

## 5.6 Discussion

**Transformation Correctness.** PolyglotPiranha does not guarantee that the transformed code will compile, be semantically correct, or precisely reflect the developer's intent. This limitation is common to other syntax-driven code transformation tools such as [23, 75, 159]. While our dataflow analysis verifies the rule graph's consistency and grammatical accuracy (Section 5.4.1.A), the effectiveness and accuracy of transformations ultimately rely on the quality of the rule graph itself.

**Syntactic Limitations.** PolyglotPiranha's purely syntactic approach limits its ability to perform transformations that require semantic information of the code. In practice, this means that code rewrites that require type resolution, class hierarchy analysis, and/or control-flow analysis can not be expressed in the DSL today. Specifically, PolyglotPiranha: 1. *lacks precise def-use information.* We designed rules conservatively to identify def-use relationships within the syntactic scope of the variable declaration. However, due to the lack of SSA representation and dominator information PolyglotPiranha cannot reason about variable shadowing or re-initialization. 2. *lacks precise type information.* We approximate type information by analyzing declarations within a scope. This falls short when dealing with language features that obscure type information, such as

`Java`'s `var` keyword or dynamically typed languages like `Python`. 3. *lacks call-graph analysis.* We approximate caller-callee relationships using method names and their number of arguments, resulting in imprecision in the presence of interfaces, class hierarchies, and method overloading. 4. *cannot handle advanced language features* that require semantic analysis, such as reflection.

Despite these limitations, our evaluation showcases that PolyglotPiranha is effective at automating three real-world code transformation tasks. Though imperfect, even in cases where it was partial, this automation substantially alleviated developers' load as seen in Figure 5.11a.

**Supporting New Languages.** PolyglotPiranha supports languages beyond the ones listed in the evaluation, including `Go`, `Python`, `Scala`, `Typescript`, as well as protocol formats like `Thrift`. PolyglotPiranha uses `tree-sitter` for code parsing, thus supporting a new language requires: 1. incorporating the `tree-sitter` grammar within PolyglotPiranha, and 2. authoring scope-capturing rules in a configuration file (i.e., one rule per scope such as class, method, or file). PolyglotPiranha uses these scopes when applying rules from the rule graph. Note that `tree-sitter` officially supports 133 programming languages [93], including functional languages like `Haskell` and `Scheme`. In fact, we support `Scheme` as a language in PolyglotPiranha, and use it within PolyglotPiranha's implementation for rewriting its structural queries (a subset of `Scheme`). The implementation burden for this support was comparable to other languages.

Adapting PolyglotPiranha-based tools, like those for feature flag cleanup, to new languages may require additional work. For example, a rule for simplifying a disjunction `(true || :[a])` in `Java` needs to be customized for `Python` as `true` or `:[a]`. However, we observed that some rules are reusable within a broad family of languages (`Java`, `Kotlin`, etc).

**PolyglotPiranha's Usability.** To assist users in debugging and root-causing failures due to errors in the rule graph, PolyglotPiranha outputs detailed reports of all executed rules (in order) including their corresponding matched `LoC` ranges, and runtime arguments in an easily queryable format. This allows for step-by-step replay and analysis. Our repository contains examples that explain how to enable debugging mode. We have also developed a playground for rule experimentation that allows users to easily experiment with rules and rules graphs on code snippets. This playground is publicly available on our artifact.

## 5.7 Key Takeaways and Contributions

In this chapter, I introduced a novel code transformation language, PolyglotPiranha. The language and toolset were designed to support complex code transformations. We demonstrated desirable properties of the language (namely, its expressiveness, usefulness, and run-time efficiency) through three case studies. By construction, this language is more expressive than `comby`, while maintaining its lightweight and declarative nature. The goal was to enable complex code transformation to be expressed in a lightweight declarative

language, rather than resorting to language-specific and imperative toolsets for code manipulation. In the next chapter, I show how to leverage LLMs for synthesizing complex API migration scripts in this language.

# 6

# Distilling code migration knowledge from Large Language Models

## Contents



In this chapter, I present my work on SPELL, a technique for synthesizing code migration scripts using large language models. This work is motivated by the observation that LLMs encode latent knowledge about API mappings and their knowledge that can be extracted and systematized into reusable transformation scripts. Whereas prior approaches to library migration depend on manually curated data or mined repository histories (as discussed in Chapter 2), SPELL leverages the generative capabilities of LLMs to create synthetic migration

examples and validate them. It also overcomes the limitations of SOAR related to runtime performance discussed in Chapter 3 by representing the programs in the PolyglotPiranha language. This chapter introduces the full pipeline underlying Spell, from LLM-guided data generation to agent-based script synthesis in the PolyglotPiranha language. Finally, the tooling is evaluated across a diverse set of libraries in python.

## 6.1 Motivation

As described in previous chapters, most existing methods rely on mining software repositories that have already undergone migration [18–22, 44, 143, 175], which are then generalized and systematized into reusable formats for broader application.

However, these techniques are limited due to their dependency on the availability and quality of historical migration data. First, migration data is scarce [97]. Collecting projects that have migrated between arbitrary pairs of libraries is challenging due to the combinatorial nature of the problem. Without this data, existing tools cannot infer or generate migration scripts.

Second, most existing approaches rely on custom program transformation toolsets and do not leverage available infrastructure for expressing migration logic (e.g., [19–22]). However, in recent years, several automated code transformation toolsets have emerged to support migration tasks, including tools like OpenRewrite [92], GritQL [176], and PolyglotPiranha [5]. These are domain-specific languages designed to manipulate source code, and have been used for performing automated migrations.

Recently, large language models (LLMs) have also gained traction in automating various code transformation tasks, including in industrial settings (e.g., Amazon Q Transform [118]). However, directly using LLMs to migrate or refactor code is often unreliable: models frequently introduce formatting changes, stylistic edits, or subtle semantic bugs [177]. These migrations are also inefficient, since each one is generated from scratch and produces no reusable artifact.

Nonetheless, despite these limitations, LLMs hold significant potential for the library migration domain. Through pretraining on large-scale code corpora, LLMs implicitly learn joint representations of APIs and the semantic relationships between them, including cross-library mappings [40]. While this knowledge may be imperfect, our key insight is that it can be extracted and tested to produce meaningful migration examples for learning migration scripts.

Thus, we introduce Spell (**S**ynthesis of **P**rogrammatic **E**dits using **LL**Ms), a novel approach to automated code migration that leverages this key insight. Our method prompts models to generate equivalent implementations for both libraries, along with tests to verify behavioral equivalence. By generating and filtering multiple such examples, we systematically distill migration knowledge from LLMs and encode it into reusable, testable scripts in a source code transformation language. Unlike prior approaches, Spell does not depend on existing migration histories or manually crafted transformation rules — though it implicitly draws on the

**(a)** Code snippet using `cryptography` (before)

```python
import hashlib
from cryptography.fernet import Fernet

(...)

def encrypt_document(document: str, key: bytes) ->
    bytes:
    cipher = Fernet(key)
    encrypted = cipher.encrypt(document.encode())
    return encrypted

(...)
```

**(b)** Code snippet using `pycryptodome` (after)

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad

(...)

def encrypt_document(document: str, key: bytes) ->
    bytes:
    cipher = AES.new(key, AES.MODE_CBC)
    padded_data = pad(document.encode(),
        AES.block_size)
    encrypted = iv + cipher.encrypt(padded_data)
    return encrypted
```

**Figure 6.1:** Simplified function-level encryption logic before (using `cryptography`) and after (using `pycryptodome`), illustrating the shift from Fernet's built-in encryption to manual AES-CBC with padding and IV management.


**Figure 6.2:** Program in the PolyglotPiranha language generated from the pair of functions presented in Figure 6.1.

LLM's training data. Rather than mining examples from real-world repositories, it produces them through a structured prompting and validation process, enabling migrations even when curated examples are unavailable.

After converting the unstructured knowledge of LLMs into concrete before-and-after code pairs, we use them to synthesize generalized transformation scripts in the PolyglotPiranha language, using an agent workflow. This allows us to produce reusable, testable scripts without manual intervention or large mined datasets.

Our results show that Spell can synthesize correct, reusable migration scripts across a range of Python library migrations. For nine tasks, it generated an average of 99 validated examples per task and inferred correct scripts for 60.9% in a single trial. Compared to `MELT`, Spell achieved higher success rates on every task. Moreover, the scripts generalized to real-world projects: when applied to GitHub repositories, they performed correct rewrites and preserved test behavior in most cases. These findings show that structured migration knowledge can be distilled from LLMs and converted into reusable migration scripts.

## 6.2 Illustrative Example

Our pipeline consists of two phases: (1) data generation, where we attempt to generate code migration pairs and tests, and (2) synthesis, where we abstract those pairs into reusable migration scripts.

### 6.2.1 Data Generation

The goal of the data generation phase is to distill code migration knowledge from the LLM in the form of migration examples.

As an illustrative example, consider the task of migrating between two Python cryptography libraries: `cryptography` and `pycryptodome`. Given a pair of libraries *(source, target)*, our approach uses an LLM to generate two implementations of the same functionality, one using the source library and the other using the target. These paired implementations form the basis for migration examples that can later be generalized.

To enable this, the first step is to define a task to be implemented. We begin by asking the LLM to generate ideas, i.e., tasks or features that could plausibly be implemented using either library. These ideas help focus the model on a shared functional goal and can be realized with both the source and target APIs. For example, when asked to propose such an idea for `cryptography` and `pycryptodome`, `gpt4o-mini` responds:

> *A **File Encrypting Utility** can be implemented using `pycryptodome` or `cryptography` to allow users to encrypt and decrypt textual documents. High-level behaviour and API: (...)*

Using this idea, we then prompt the model to generate an implementation using the source library. For example, Figure 6.1a shows one such implementation generated by the LLM using the `cryptography` library.

Subsequently, we also prompt the LLM to generate tests for each implementation. These tests help verify that the code is functional and are used to filter out examples that do not behave as expected.

Finally, we prompt the LLM to migrate each implementation in `cryptography` to the alternative library `pycryptodome`. Figure 6.1b shows the migration the LLM generated for the code in Figure 6.1a.

During migration, our prompt instructs the LLM to modify only the library-specific API calls while keeping the rest of the code unchanged. This is important because we reuse the tests generated for the original implementation and run them on the migrated version. The test results serve as a signal that the migration was successful, that is, the migrated code behaves the same as the original on the tested inputs. The tests act as a filter to discard examples that are not equivalent under test.

At the end of this process, we obtain triples: a source implementation using the source library, a corresponding target migration, and a shared test file that both the implementation and the migration successfully pass. Our data generation pipeline is described in more detail in Section 6.3.

### 6.2.2 Synthesizing Migration Scripts

The synthetic code migration pairs from the previous step are then abstracted into reusable migration programs. These programs capture structural transformation patterns, allowing the same migration logic to be applied across similar tasks.

We express these programs in the PolyglotPiranha [5] language, our DSL for source-to-source transformations. Recall from Chapter 5 that a PolyglotPiranha program is composed of `rules`, each with a `match`

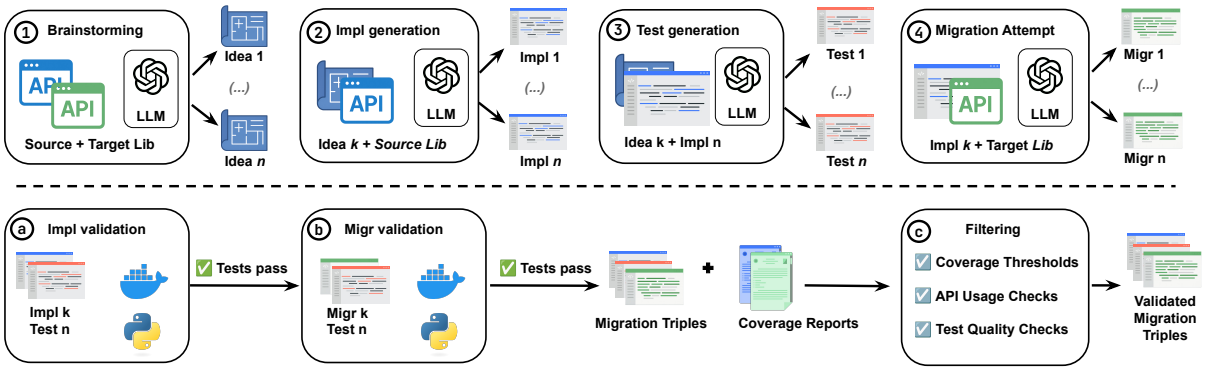



**Figure 6.3:** Overview of our data extraction pipeline for generating migration examples using LLMs. Given a source–target library pair, we first prompt the LLM to propose abstract ideas leveraging its latent knowledge of both libraries (Step 1). For each idea, we generate multiple implementations using the source library (Step 2), followed by corresponding test cases (Step 3). We then attempt to migrate these implementations to the target library (Step 4). The generated data undergoes a three-stage validation process: implementation validation (a), migration validation (b), and quality filtering based on coverage, API usage, and test robustness (c). The final output is a collection of validated migration triples (implementation, test, migration).

clause that identifies a code pattern and a `replace` clause that rewrites it. Template variables (e.g., `:[var]`) allow rules to generalize over specific names and expressions, effectively acting like regular expressions, but for syntax-aware code patterns. Rules can be connected through labeled edges that specify how and where follow-up transformations should be triggered. This enables cascading and context-sensitive rewrites.

Figure 6.2 shows the PolyglotPiranha program synthesized from the pair of functions in Figure 6.1. The first rule, `replace import`, replaces the original import with one from the target library. This triggers the second rule within the same file, `replace decl`, which rewrites the construction of the object. Finally, the third rule, `replace encrypt`, updates how the encryption API is used. The template variables like `:[var]` and `:[data]` ensure that these rules are not tied to specific variable names or concrete expressions.

In Section 6.4, we describe how our agent workflow automatically synthesizes these rules from the migration triples. The output of this step is a set of interpretable migration programs that capture the essence of the transformation and can be adapted or extended to similar scenarios.

## 6.3 Migration Data Extraction

Figure 6.3 overviews the data distillation process, which extracts and validates LLM's pre-trained knowledge about migrating between two libraries. The process generates sets of equivalent programs using source and target libraries, with test cases that provide evidence of their functional equivalence. This data is the input for migration script synthesis (Section 6.4).

More precisely: given source library $\mathcal{S}$ and target library $\mathcal{T}$, we sample migration knowledge from an LLM as raw migration triples $\widetilde{\mathcal{M}} = \{(\tilde{s}_i, \tilde{t}_i, \tilde{m}_i)\}_{i=1}^{n}$. These triples are automatically generated and may contain redundancy, inconsistencies, or errors. We validate and filter this set to produce a high-quality subset $\mathcal{M} =$

$(s_i, t_i, m_i)_{i=1}^{m}$, where $m \leq n$, and:

- $s_i$ is an implementation using $\mathcal{S}$
- $m_i$ is a functionally equivalent implementation using $\mathcal{T}$
- $t_i$ is a test suite that executes and passes on both $s_i$ and $m_i$

We use tilde to denote sampled triples that may or may not be valid; notation without a tilde refers to data/triples that have been validated. We now explain knowledge extraction (Section 6.3.1) and validation (Section 6.3.2).

### 6.3.1 Model Sampling

The goal in model sampling is to generate a diverse initial set of example implementations of shared functionality, using $\mathcal{S}$ and $\mathcal{T}$. A nontrivial library can usually be used in a variety of ways. Thus, our process entails a multi-stage pipeline that first generates multiple ideas of functionality that can be implemented using $\mathcal{S}$ (Section 6.3.1.A), and separately generating implementations of those ideas using $\mathcal{S}$ (Section 6.3.1.B). For each implementation of each idea, the pipeline generates tests (Section 6.3.1.C), and finally migrates the implementation to use $\mathcal{T}$ (Section 6.3.1.D).

#### 6.3.1.A Idea Generation

Given libraries $(\mathcal{S}, \mathcal{T})$, SPELL first prompts an LLM to generate multiple ideas $\mathcal{I}$ (Figure 6.3, Step 1). Each idea $I_j \in \mathcal{I}$ represents a use case, program, or otherwise useful functionality that could be implemented using either library.

Generating ideas first decouples the creative task of brainstorming migration scenarios from the technical task of writing correct code. This serves to encourage a model to explore a wide space of API usages, beyond common boilerplate. It also provides a clear, shared semantic goal, encouraging consistency between source and target implementations. Finally, it allows the pipeline to retry implementation generation for promising ideas as necessary, without having to re-explore the entire conceptual space. Decoupling also allows the model to focus on brainstorming ideas that are not complex or otherwise unsuitable, before investing computational resources in their implementation, next.

Idea generation begins by using the model to create a small set of seed ideas that are then used for a self-instruct loop to generate additional migration scenarios. Self-instruction, in which a model produces its own task demonstrations using seed data, has proven effective in bootstrapping a model's existing knowledge in a variety of contexts [178]. We first generate $p$ seed ideas by prompting the model directly with a basic instruction. Each idea describes a use case involving a source library, along with a high-level summary of its functionality. These seed ideas are collected into a list and used as few-shot examples in subsequent prompts to generate more.

Prompt for Idea Generation

Generate a functionality that a developer might commonly implement using either `{library_name_1}` or `{library_name_2}` in `python`. Focus on operations that are well-suited to these libraries' strengths and typical use cases. Describe the high-level behavior and expected outcomes, without diving into specific implementation details. The functionality should be simple enough. It should be easy to implement in at most 100ish LoC.

**Figure 6.4:** Prompt SPELL uses for generating different ideas to implement in two libraries of interest.

Prompt for Implementation

Implement the functionality described above using the `{library_name}` library. Follow best practices for this library, ensuring that the implementation is modular, clear, and easy to test. The implementation should focus on achieving the high-level behavior described, while abstracting away unnecessary complexity. Design functions that can be easily be replaceable by alternative implementations.

Guidelines for your implementation:

- **Do NOT** use classes, unless absolutely necessary. Please use functions instead.
- You may include a small `main` block, but please focus on API implementation.
- Everything needs to be self-contained in this `python` file!

**Figure 6.5:** Prompt SPELL uses for eliciting concrete implementations from generated ideas.

The idea generation loop proceeds by randomly sampling $k$ prior ideas (seed or generated), to include in a new prompt, and asking the model to propose one new idea. This repeats until it produces a target number of ideas. This step outputs a set of $n$ ideas that demonstrate ways the source and target libraries can be used.

#### 6.3.1.B Implementation Generation

For each idea $I_j \in \mathcal{I}$, we prompt the model to generate a set of implementations $\widetilde{S}_j = \{\tilde{s}_{j,1}, \tilde{s}_{j,2}, ..., \tilde{s}_{j,n}\}$ using source library $\mathcal{S}$ (Figure 6.3, Step 2). Ideally, these implementations are modular, with a well-defined API. This simplifies migration to $\mathcal{T}$ while preserving the high-level interface, necessary to use the same tests on both.

The prompt thus instructs the model to implement the idea in a self-contained `python` file, subject to the following constraints: 1. implement the API for idea $I_j$ as a set of functions, 2. focus on API implementation; a small `main` block is optional, and 3. avoid external dependencies beyond the specified library $\mathcal{S}$.

We construct a multi-turn chat prompt that includes the original idea and implementation instructions. To obtain diverse outputs and increase the chances that the model produces usable implementations, we sample multiple completions using stochastic decoding.

The output of this step is a set of up to $n$ potential implementations of each $I_j \in \mathcal{I}$.

Prompt for Integration Tests

Now, I would like you to write some tests. Please abstract from concrete implementation details and focus on testing the actual functionality without relying on mocks or other implementation-specific constructs (think of an integration test).

Write a set of integration tests for your implementation using either `pytest` or `hypothesis`. The tests should focus on verifying the correctness of the functionality across a wide range of inputs, including edge cases. Ensure that the tests are comprehensive and emphasize high-level behavior and expected outcomes, rather than specific implementation details.

The tests should be resilient to changes in the underlying implementation, ensuring they still pass if the API or library is replaced with an alternative.

Assume the file you generated will be under the same directory `./implementation.py` for import purposes. Use relative imports to import the implementation file.

**Figure 6.6:** Prompt SPELL uses to generate integration tests for a previously implemented idea.

#### 6.3.1.C Test Generation

For each idea-implementation pair $(I_j, \tilde{s}_{j,k})$, we generate multiple test suites $\widetilde{T}_{j,k} = \{\tilde{t}_{j,k,1}, \tilde{t}_{j,k,2}, ..., \tilde{t}_{j,k,p}\}$ (Figure 6.3, Step 3). A test suite $T_{j,k}$ is intended to partially verify, and build confidence in, the correctness of implementation $s_{j,k}$ (and, later, the migrated implementation using target library $\mathcal{T}$).

The ideal tests target an implementation's public API and core functionality, rather than internal implementation details that may not transfer. For example, tests for a program that encrypts files should not inspect the encrypted output directly, but should instead verify that decrypting a file restores the original content (among other functionality). The prompt instructs the model to generate unit tests (or end-to-end tests, if unit tests are infeasible) that validate the overall system behavior.

The prompt includes $I_j$, $\tilde{s}_{j,k}$, and the test generation instructions. We sample multiple test files per implementation to mitigate model hallucination as well as challenges in test generation generally, such as incorrect assertions. Sampling increases the chances of obtaining at least one valid test file for an otherwise correct implementation. Since our implementation is `python`-specific, the prompt instructs the model to use either the `pytest` testing framework or property-based testing with `hypothesis`, when appropriate. This assumption could be easily ported to other languages.

The output of this step is multiple test files per implementation.

#### 6.3.1.D Migration

For each idea-implementation pair $(I_j, \tilde{s}_{j,k})$, we prompt the model to generate multiple migrated implementations $\widetilde{M}_{j,k} = \{\tilde{m}_{j,k,1}, \tilde{m}_{j,k,2}, ..., \tilde{m}_{j,k,q}\}$ in target library $\mathcal{T}$ (Figure 6.3, Step 4). Each $m_{j,k,r}$ represents an attempt to migrate $s_{j,k}$ while preserving its public API. This entails a multi-turn chat prompt that includes the original

Prompt for Library Migration

Migrate the following implementation from `{original_library_name}` to `{alternative_library_name}`: Please maintain the same API. Keep the same function names, arguments, and class names!! The migrated code should pass the original integration tests, confirming consistent behavior across both libraries.

Ensure that the new implementation adheres to the same modular design and high-level behavior as the original. It should keep the API names, class names, etc. The migrated code should pass the original integration tests, confirming consistent behavior across both libraries. Replace any references to `{original_library_name}` with `{alternative_library_name}` in the code. Please generate the entire file except with this particular function migrated to the new library.

**Figure 6.7:** Prompt SPELL uses to perform a library migration while preserving the original API and integration behavior.

idea $I_j$ and implementation $\tilde{s}_{j,k}$, and instructions asking the model to migrate the source implementation to use the target library. The prompt further instructs the model to keep the same structure and function signatures, and only update the internal logic and API calls to use the target library.

The output of this step is the set of raw migration triples $\widetilde{\mathcal{M}} = \bigcup_{j,k}\{\tilde{s}_{j,k}\} \times \widetilde{T}_{j,k} \times \widetilde{M}_{j,k}$, where the union is taken over all ideas $I_j \in \mathcal{I}$ and their implementations. These may include incorrect or only partially migrated examples, so they are validated and filtered as described in Section 6.3.2.

### 6.3.2 Validation

Not all sampled migration examples are valid. Initial or migrated implementations may be spurious or incomplete, tests may not run, etc. Thus, the validation step filters $\widetilde{\mathcal{M}}$ to produce a set of valid migration triples $\mathcal{M} = \{(s_i, t_i, m_i)\}_{i=1}^{m}$. In a valid triple, both source and migrated implementations expose desired APIs, pass the same tests, and are exercised with sufficient test coverage.

**Implementation Validation.** First, we validate initial idea implementations. For each $(\tilde{s}_{j,k}, \tilde{t}_{j,k,\ell}) \in \widetilde{S}_j \times \widetilde{T}_{j,k}$, we execute tests $\widetilde{T}_{j,k}$ against $\widetilde{s}_{j,k}$, retaining pairs where all tests pass. Recall that we generate multiple potential test suites per implementation (Section 6.3.1.C; this step can therefore produce multiple pairs containing a given $s_{j,k}$. We run each test file $\tilde{t}_{j,k,\ell}$ independently using `pytest` inside a `docker` container that includes a pinned `python` version and a large set of `python` libraries, including $\mathcal{S}$ and its dependencies.

**Migration Validation.** For each valid $(s_{j,k}, t_{j,k,\ell})$ pair and associated migration attempt $\tilde{m}_{j,k,r} \in \widetilde{M}_{j,k}$, we improve confidence in equivalence by confirming that $\tilde{m}_{j,k,r}$ also passes tests $t_{j,k,\ell}$ (dropping those that do not). Because the migrated code is expected to preserve the original API, the same test suite is expected to apply (and those that do not, are dropped).

To construct the final set $\mathcal{M}$, we select a single triple per implementation. Specifically, for each implementation $\tilde{s}_{j,k}$ with multiple valid test–migration pairs, we retain the triple $(s_i, t_i, m_i)$ with the highest post-

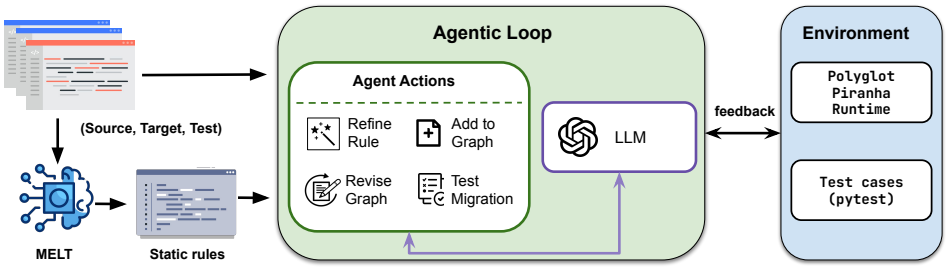



**Figure 6.8:** Agentic workflow of SPELL's approach to migration script synthesis. Our pipeline leverages MELT for creating an initial set of rules, which is then fed to an LLM agent for POLYGLOTPIRANHA script generation.

migration coverage. This ensures that each source program maps to exactly one migration.

**Test-based Filtering.** Finally, we also collect line-level coverage metrics for each run to build confidence that the test suites meaningfully exercise core logic. We simultaneously collect information on API usage, for later analysis. We retain triples where shared tests achieve at least 60% line coverage on both implementations.

To construct the final validated $\mathcal{M}$: for each implementation $\tilde{s}_{j,k}$ of idea $I_j$ part of more than one valid triple, we select the one with the highest post-migration test coverage. We include implementations part of only a single valid triple directly. These selected triples become $(s_i, t_i, m_i) \in \mathcal{M}$. Thus, each validated implementation contributes at most one triple to $\mathcal{M}$, ensuring high-quality examples while maintaining idea implementation diversity.

## 6.4 Migration Script Synthesis

SPELL converts migration triples $\mathcal{M} = \{(s_i, t_i, m_i)\}_{i=1}^{n}$ (Section 6.3) into executable rewrite scripts in POLYGLOTPIRANHA, a state-of-the-art DSL and toolset for code transformation (Section 5.1). Our novel approach for script inference combines classical anti-unification with an agentic LLMs strategy. This hybrid is motivated by the structure of POLYGLOTPIRANHA programs: local rewrite rules (graph nodes) connected by a high-level strategy (labeled edges). Classical anti-unification excels at inferring precise local rules from concrete examples, but cannot infer the orchestration strategy, scope, or cascading effects. We therefore complement the traditional synthesis with an LLM-based technique to organize mechanically-inferred atomic rules into a coherent transformation strategy.

For each migration triple $(s, t, m) \in \mathcal{M}$ script synthesis entails: (1) **Rule Inference**: Automatically generate initial low-level transformation rules from source-target diff hunks (Section 6.4.2), and (2) **Agent-based Orchestration**: Use LLMs to refine and orchestrate the inferred rules into a rule graph (Section 6.4.3). This overall produces a migration program per successfully-processed triple in $\mathcal{M}$.

### 6.4.1 PolyglotPiranha

We infer migration scripts in PolyglotPiranha, a domain-specific language for source-to-source introduced in Section 5.1. PolyglotPiranha exemplifies a growing class of modern transformation DSLs [75, 92, 176] designed to support industrial-scale automated refactoring. These tools offer a structured, declarative alternative to ad hoc migration scripts.

Recall some of the key language constructs introduced in Section 5.1: PolyglotPiranha programs consist of local syntactic rewrite rules organized into a directed graph encoding application order, scoping constraints, and inter-rule dependencies. Each node defines a rule with a match and an optional rewrite clause; edges model control flow using scoped labels. I.e., an edge $\mathcal{R}_1 \xrightarrow{\text{Function}} \mathcal{R}_2$ reads as, "apply rule $\mathcal{R}_1$ and then apply rule $\mathcal{R}_2$ within the enclosing function where $\mathcal{R}_1$ was applied". Rules themselves are written as concrete syntax patterns (that closely resemble the target language's surface syntax). The concrete patterns are composed of strings and abstractions that match directly against concrete code. These abstractions are expressed via template variables (e.g., `:[x]`), which bind to concrete sub-expressions. For example, the rule `foo(:[args+])` $\mapsto$ `bar(:[args])` rewrites `foo(1,2,3)` as `bar(1,2,3)`, where `:[args]` binds to `(1,2,3)` during the transformation.

PolyglotPiranha programs run as depth-first traversal of the rule graph. The runtime begins with a queue of seed global rules and applies them depth-first until reaching a fixpoint. We refer interested readers to associated materials and documentation for full syntax and semantics [179].

Thus, for each migration triple $(s, t, m) \in \mathcal{M}$, we aim to synthesize a PolyglotPiranha script $\mathcal{P} = (\mathcal{R}, \mathcal{E})$ where:

- $\mathcal{R} = \{r_1, r_2, ..., r_k\}$ is a set of transformation rules
- $\mathcal{E} \subseteq \mathcal{R} \times \mathcal{L} \times \mathcal{R}$ is a set of directed edges with labels from $\mathcal{L} = \{\text{File, Function, Class, ...}\}$

Each rule $r_i \in \mathcal{R}$ is a tuple $(p_i, q_i)$ where $p_i$ is a match pattern, and $q_i$ is a replacement pattern. An edge $(r_i, \ell, r_j) \in \mathcal{E}$ indicates that after applying rule $r_i$, rule $r_j$ should be applied within scope $\ell$.

### 6.4.2 Atomic Rule Inference

Let $\Delta(s, m)$ denote the set of diff hunks between source implementation $s$ and migration $m$. For each diff hunk $h \in \Delta(s, m)$, let $h^-$ and $h^+$ denote the deleted and added lines, respectively.

We use an anti-unification algorithm to abstract these hunks into reusable rewrite rules. Anti-unification computes the most specific generalization of two expressions—deriving a pattern that captures common structure while abstracting over differences using placeholders—making it ideal for synthesizing migration rules that preserve structural essence while enabling generalization. We adapt an existing anti-unification

algorithm $\mathcal{A}$ (from MELT [45]) to infer an initial rule set:

$$\mathcal{R}_0 = \{r : r = \mathcal{A}(h^-, h^+) \text{ for } h \in \Delta(s, m) \text{ where } h^- \neq \emptyset\}$$

. Each rule abstracts shared elements in the *before* and *after* hunk.

For example, given the following hunk from a larger migration of `cryptography` to `pycryptodome`:

```
- encrypted_data = fernet.encrypt(data)
+ padded_data = pad(data, AES.block_size)
+ encrypted_data = iv + fernet.encrypt(padded_data)
```

Anti-unification, including heuristics to avoid overgeneralizing, produces: `:[x1]= :[x2].encrypt(:[x3])`

```
  ↦ padded_data = pad(:[x3], AES.block_size)
    :[x1]= iv + :[x2].encrypt(padded_data)
```

Note that the algorithm we adapt does not support rule generation for hunks in which code is only added without deletion [45]; we discuss this limitation in Section 4.4.

### 6.4.3 Script synthesis

Rule inference provides initial ruleset $\mathcal{R}_0$, but no composition strategy. Agentic script synthesis constructs the full script $\mathcal{P} = (\mathcal{R}, \mathcal{E})$ through iterative refinement. This refinement also allows SPELL to correct initial rules $r$ that overgeneralize or are incorrect. An agentic approach for software tasks uses an LLM to iteratively take action, and observe its effects to refine a strategy for decomposing and solving a complex problem.

Figure 6.8 illustrates the workflow. It takes as input a migration triple $(s, t, m) \in \mathcal{M}$ and the initial rule set $\mathcal{R}_0$ inferred from diff $\Delta(s, m)$ between the source and target implementations. The model's task is to develop the overarching strategy comprising the full script while disambiguating conflicting rules and deciding on scopes and application order; and improving the transformation rules as necessary by, for example, resolving naming problems. The rest of this Section details key components of the approach.

**System prompt.** We assume a model is unfamiliar with the relatively new POLYGLOTPIRANHA language. The system prompt explains language syntax, semantics, and runtime behavior, via: (1) examples of simple rules, explanations of concrete syntax, and guidance on matching/rewriting; (2) an explanation of rule graph, rule application strategies, scoping, and constructs for propagating template variables across rules; (3) examples of source code before and after migration, with detailed explanations of how POLYGLOTPIRANHA performs migration, and runtime traces, and finally (4) other guidelines, common pitfalls, and best practices.

**Agentic Loop.** For each migration triple $\mathcal{M} = \{(s_i, t_i, m_i)\}$ and initial ruleset, SPELL's agentic loop iteratively takes actions and observes their effects until it produces a POLYGLOTPIRANHA program that transforms the source $s_i$ into a migration $\hat{m}_i$ that passes the original test suite $t_i$, or until it completes 10 iterations.

The first iteration takes a partially-complete task template containing the migration triple and initial SPELL-inferred rules. We prompt the model to generate a POLYGLOTPIRANHA program that transforms $s_i$ into $m_i$, following system guidelines. The prompt specifies that the produced program need not produce an exact token-level match with $m_i$, but should capture the intended semantic transformation. We (and the agent) verify correctness by running code produced by the eventual program on test suite $t_i$.

Each iteration, the model selects an action and executes it in a controlled environment, with result returned as feedback:

- **Refine / Create an Atomic Rule.** The model can create a new match-replace rule, or refine an existing one, and apply it to $s_i$ in isolation to observe an atomic rule's behavior independently of other transformations.
- **Add a Rule to the Graph.** The model can add one or more new rules to the (initially empty). The action specifies rule order and scope.
- **Revise the Current Rule Graph.** A model can revise the graph beyond atomic rule refinement by modifying portions of the graph, or regenerating it, enabling high-level restructuring of the transformation strategy.
- **Test Migration.** The model can use tests $t_i$ to validate the current rule graph by applying it to $s_i$ and running tests $t_i$ against the result. If the tests pass, the loop terminates.

**Environment.** SPELL executes actions within a structured environment and provides results back to the model to guide the next agentic step. For rule refinement and integration, and rule graph revision, SPELL applies generated $P$ to $s_i$ using the POLYGLOTPIRANHA interpreter and returns the resulting migrated program as feedback. SPELL provides additional feedback when available, e.g., if applying $P$ fails due to a syntax or POLYGLOTPIRANHA runtime error, SPELL returns the error message with a set of likely causes and correction suggestions. These are derived from common failure modes and known pitfalls in the language engine, akin to compiler diagnostic messages. If the script executes but does not change $s_i$, SPELL provides a set of potential reasons, again with common pitfalls. For testing, SPELL applies the script and runs the transformed code inside a containerized test environment. If all tests pass, SPELL records the migration as successful; Otherwise, it returns the test failures and error messages for the next iteration.

## 6.5 Evaluation

We answer four research questions:

**RQ1. (End-to-end-effectiveness): How effectively does SPELL generate POLYGLOTPIRANHA migration scripts?**
We evaluate the extent to which the entire pipeline produces working POLYGLOTPIRANHA scripts.

**Table 6.1:** Overview of migration tasks with synthesis results. Valid Triples: generated migration triples with >60% coverage. Success: percentage of migration triples for which each tool successfully generated a validated PolyglotPiranha script. Sibling Success: sibling implementations that at least one synthesized PolyglotPiranha script migrates successfully out of the total number of potentially-migratable sibling implementations.

| Migration Task | | | | Valid | Success (%) | | Sibling |
|---|---|---|---|---|---|---|---|
| Source | Target | Domain | Stars (S / T) | Triples | Spell | MELT | Success |
| argparse | click | CLI | N/A / 16.4k | 215 | 44.2 | 17.2 | 91/170 |
| jinja2 | mako | Templating engines | 10.9k / 389 | 15 | 13.3 | 0.0 | 0/3 |
| json | orjson | JSON Serialization | N/A / 6.9k | 269 | 96.7 | 57.6 | 258/291 |
| logging | loguru | Logging | N/A / 21.7k | 114 | 85.1 | 72.8 | 64/93 |
| lxml | BeautifulSoup | HTML/XML parsing | 2.8k / N/A | 32 | 59.4 | 0.0 | 0/21 |
| pandas | polars | Data analysis | 45.5k / 33.8k | 0 | - | - | - |
| pathlib | pyfilesystem2 | Filesystem abstraction | N/A / 2.0k | 21 | 52.4 | 28.6 | 1/15 |
| cryptography | pycryptodome | Cryptography | 7.0k / 3.0k | 79 | 48.1 | 6.3 | 3/46 |
| requests | httpx | HTTP clients (async, sync) | 52.9k / 14.1k | 65 | 76.9 | 12.3 | 39/63 |
| time | pendulum | Date/time | N/A / 6.4k | 60 | 78.3 | 11.7 | 35/73 |

**RQ2. (Real-world applicability): How useful are Spell's scripts for migrating real-world client code?** We test whether the Spell-produced PolyglotPiranha scripts, learned from synthetic examples, successfully migrate real open-source projects, to evaluate practical utility.

**RQ3. (Comparison with prior work): How does Spell compare to MELT, a prior tool for automated refactoring?** We compare Spell to MELT, the most-closely-comparable prior approach for migration rule inference in Python.

**RQ4. (Data quality): What is the quality of the synthetic migration examples from the data generation pipeline?** A core novelty of Spell is its bootstrapping via synthetic examples. We assess the extent to which these examples are valid and diverse, and provide sufficient coverage of both source and target APIs and implementations.

### 6.5.1 Experimental Setup

**Implementation.** We implement Spell in python. Both synthetic data generation and agentic synthesis are model-agnostic and use the OpenAI Chat Completion API [180] (adopted by most major LLM API providers). For data generation, we use gpt-4o-mini (a small model) due to its cost effectiveness, drawing on prior results suggesting that more attempts offset the limitations of smaller models [181]. For synthesis, we use gpt-4.1 [182], a current state-of-the-art model with more advanced capabilities and larger context window (1M tokens), required for our large system prompt and chained interactions.

**Migration tasks.** We evaluate on 10 migration tasks across 20 python libraries; the left-hand-side of Table 6.1 summarizes. These tasks span a diverse range of popular libraries and their alternatives across multiple domains, which prior work has also targeted [44, 45, 116, 145]. We focus on python because there is a well-documented lack of refactoring tools for python [27], despite the fact that it is among the most popular programming languages [147]; this also makes it a good target for LLM-based tooling [183].

**Settings.** We generate 100 ideas per migration task $(\mathcal{S}, \mathcal{T})$. We generated 5 source implementations $\tilde{s}_{j,k}$ for each idea $I_j \in \mathcal{I}$; 5 test tests $\tilde{t}_{j,k,\ell}$ per source implementation $s_{j,k}$; and 5 migration attempts $\tilde{m}_{j,k,\ell}$ per source implementation $\tilde{s}_{j,k}$. This initially yields 500 implementations, 2,500 test files, and 2,500 migration attempts per task; these are validated and filtered as described in Section 6.3.2. We run our synthesis pipeline on each validated triple to attempt to generate a PolyglotPiranha script $\mathcal{P} = (\mathcal{R}, \mathcal{E})$. We allow the agent up to 10 iterations to produce the script.

We chose these numbers (10, 5) to balance cost and coverage: based on model pricing, this setup allowed us to stay within a $50 total budget while still producing meaningful results. We generated more migration attempts because migration is a harder problem; multiple attempts increase the chance of success [181].

A synthesis attempt is considered successful if the script replaces all source library API calls ($\text{APIs}(\mathcal{P}(s), \mathcal{S}) = \emptyset$) and, if when applied to $s_j$, the migrated code passes the same tests as $m_{j,k}$ ($\mathcal{P}(s) \equiv_t m$).

### 6.5.2 RQ1: End-to-end effectiveness

**Methodology.** We first evaluate our synthesis pipeline end-to-end. We report, for each migration task, the number of valid triples generated over all ideas (Table 6.1, 5th column, "Valid triples"); this is the output of data generation (Section 6.3). Success rate (for Spell, Table 6.1 column 6) reports the percentage of valid examples on which script synthesis produce a successful PolyglotPiranha program. We investigate synthetic data quality and diversity in Section 6.5.5. To evaluate script generalizability, we test each $\mathcal{P}_{j,k}$ (from triple $(s_{j,k}, t_{j,k,\ell}, m_{j,k,r})$) on sibling implementations $\{s_{j,i} : i \neq k\}$ of the same idea $I_j$. We report how many siblings are successfully migrated by the generated scripts, per the tests.

**Results.** Table 6.1 shows results. Spell successfully generated valid migration triples for nine of ten tasks, producing an average of 87 filtered, valid triples per task (the `pandas → polars` exception is discussed in Section 6.5.5). Using these validated examples, Spell's migration synthesis script succeeded, on average, 61.6% of the time, using a single trial. This rate could likely improve further with test-time scaling [181]; we leave this to future work, given cost.

The last column of Table 6.1 shows that the scripts migrated 63.3% of sibling implementations (491/774, last column of Table 6.1). Performance was particularly strong for simple, one-to-one API replacements like `json → orjson` (88.6 %) while migrations involving embedded DSLs (`jinja2 → mako`, `lxml → beautiful-soup`) generalize less well. These libraries support dynamic webpage generation and embed a templating languages within `python` string literals, which the PolyglotPiranha engine cannot transform; this is a limitation of the PolyglotPiranha language itself [184]. Despite this constraint, our synthesized rules effectively capture common usage patterns across most migration tasks.

**Table 6.2:** Inferred migration scripts applied to real-world repositories. Each row represents the application of a migration script to a repository that uses the source library. **Stars** and **KLoC** refer to the GitHub popularity and codebase size. **Rewrites** is the number of times a PolyglotPiranha rule triggered in the project. **Passing Tests (Before/After)** test cases passing pre- and post- migration, a basic signal of functionality preservation.

| Migration Task | Repository | Stars | KLoC | Rewrites | Passing Tests Before | Passing Tests After |
|---|---|---|---|---|---|---|
| `json → orjson` | `BIM2SIM/bim2sim` | 57 | 33.8 | 46 | 173 | 149 |
| | `kaapana/kaapana` | 196 | 69.8 | 317 | 45 | 43 |
| | `SoCo/SoCo` | 1500 | 17.4 | 7 | 233 | 233 |
| `logging → loguru` | `fastqe/fastqe` | 172 | 0.7 | 16 | 2 | 2 |
| | `mie-lab/trackintel` | 229 | 11.2 | 14 | 411 | 411 |
| | `pywbem/pywbem` | 42 | 171.7 | 16 | 2499 | 0 |
| `lxml → BeautifulSoup` | `acl-org/acl-anthology` | 528 | 19.4 | 111 | 433 | 433 |
| | `pyxnat/pyxnat` | 73 | 7.7 | 8 | 62 | 62 |
| | `w3af/w3af` | 43 | 201.8 | 10 | 2547 | 2547 |
| `cryptography → pycryptodome` | `aomail-ai/aomail-app` | 134 | 2.0 | 1 | 13 | 11 |
| | `BM/qpylib` | 32 | 2.0 | 4 | 139 | 127 |
| | `DMcP89/harambot` | 32 | 1.6 | 8 | 8 | 0 |
| `requests → httpx` | `Kildrese/scholarBibTex` | 33 | 0.1 | 5 | 1 | 1 |
| | `wjohnson/pyapacheatlas` | 175 | 13.9 | 22 | 128 | 128 |
| | `databricks/databricks-sdk-py` | 433 | 108.6 | 23 | 157 | 157 |
| `time → pendulum` | `clld/clld` | 57 | 9.1 | 4 | 308 | 308 |
| | `fbpic/fbpic` | 192 | 25.5 | 43 | 6 | 5 |
| | `qutip/qutip` | 1800 | 41.2 | 29 | 1585 | 147 |

### 6.5.3 RQ2: Real-world applicability

**Methodology.** To evaluate whether our inferred migration scripts generalize beyond synthetic examples, we applied them to 18 Python projects hosted on GitHub. We used `Sourcegraph` [155] to collect a convenience sample of codebases containing API calls present in our generated scripts, prioritizing codebases with test suites. We manually reviewed each project to identify the most-relevant generated migration script that best matched observed API usage patterns. We containerized execution using `Dockerfiles` based on each project's documented setup, ran tests to establish baseline functionality, then applied the selected PolyglotPiranha script and re-ran tests. We updated `requirements.txt` to include target dependencies, but retained source dependencies, because our scripts do not usually cover all API usages within a library. We did not measure transformation time, as PolyglotPiranha's engine executes near-instantaneously. These non-trivial codebases typically span hundreds of files and thousands of lines of code.

**Results.** Table 6.2 reports the number of rewrites (i.e., the number of times a particular PolyglotPiranha rule was triggered to transform code) per project, as well as the number of passing tests before and after migration. In many cases, all tests continued to pass post-migration, despite the scripts being inferred from synthetic examples. For example, for the `logging → loguru` migration, the inferred script preserved test behavior in both the `trackintel` and `fastqe` projects and triggered a substantial number of rewrites. Similarly, Spell's scripts for `json → orjson` and `requests → httpx` applied dozens to hundreds of edits

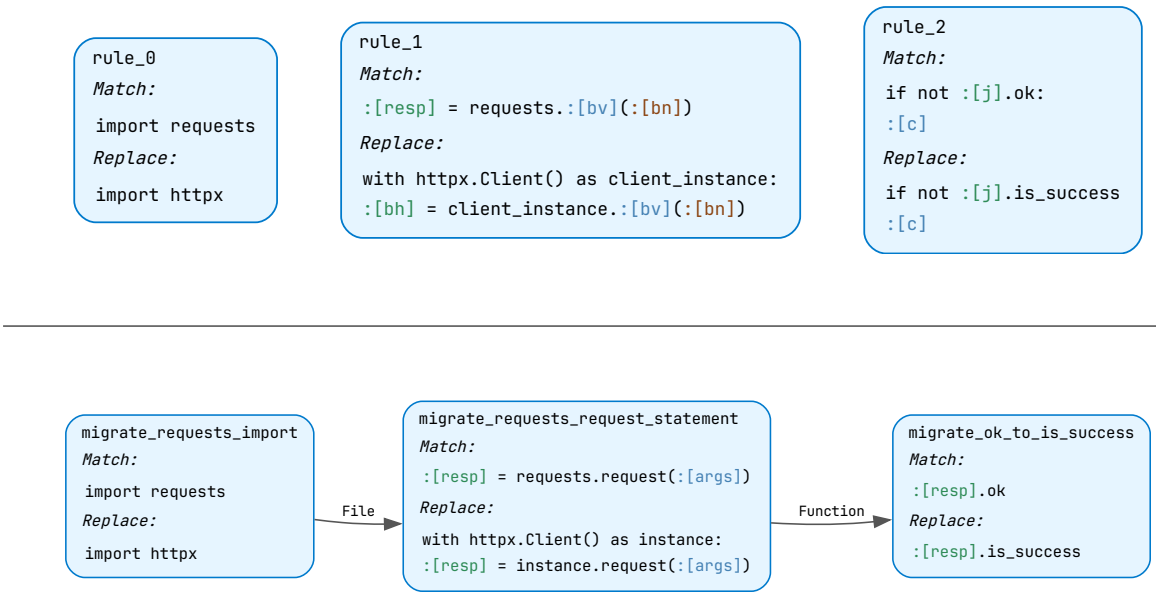



**Figure 6.9:** A program generated by MELT (top) and SPELL (bottom) from a migration example for `requests → httpx`. MELT's rules contain generic placeholder names and overabstracts; SPELL's script includes scoped application and semantically meaningful names.

across real-world projects while preserving functional behavior. Upon manual inspection, we found that test failures were typically due to incomplete migrations: the scripts handled part of the migration correctly but required further refinement or additional transformations. In other cases, particularly for `jinja2 → mako` and `lxml → beautifulsoup`, the transformations involved rewriting code inside string templates or HTML/XML fragments embedded in strings. These constructs require capabilities beyond what the POLYGLOTPIRANHA language currently supports.

### 6.5.4 RQ3: Comparison with prior work

**Methodology.** We compare our synthesis approach with MELT [45], a state-of-the-art tool for automated python refactoring. MELT because it is one of the few tools that target python and builds on well-established transformation DSL (comby [23, 140]). MELT infers synthesis scripts from migration pairs mined from developer pull requests on updated libraries, rather than generated or synthetic examples as SPELL does. To enable a fair comparison (not all of our migrations are associated with such pull requests) that more effectively isolates the hybrid synthesis in SPELL, we run MELT on the generated migration examples.

**Results.** Table 6.1, columns 6 and 7, compares the two approaches. SPELL outperforms MELT on all tasks for which valid triples are available, often by a large margin. It achieves a 61.6% (average) success rate com-

**Table 6.3:** Quality of generated migration examples: idea success rates, API diversity, and test coverage

| Migration Task | Ideas (out of 100) | | Distinct APIs | | Avg. Coverage | | Validated |
|---|---|---|---|---|---|---|---|
| | Implemented | Migrated | Source | Target | Source | Target | Examples |
| `argparse → click` | 74 | 72 | 11 | 16 | 0.89 | 0.93 | 215 |
| `jinja2 → mako` | 24 | 14 | 9 | 4 | 0.86 | 0.86 | 15 |
| `json → orjson` | 96 | 96 | 4 | 3 | 0.70 | 0.70 | 269 |
| `logging → loguru` | 58 | 52 | 12 | 6 | 0.79 | 0.79 | 114 |
| `lxml → beautifulsoup` | 59 | 23 | 11 | 9 | 0.75 | 0.75 | 32 |
| `pandas → polars` | 58 | 2 | 5 | 4 | 0.43 | 0.43 | 0 |
| `pathlib → pyfilesystem2` | 64 | 23 | 10 | 17 | 0.71 | 0.70 | 21 |
| `cryptography → pycryptodome` | 72 | 52 | 59 | 25 | 0.73 | 0.72 | 79 |
| `requests → httpx` | 64 | 60 | 8 | 10 | 0.66 | 0.64 | 65 |
| `time → pendulum` | 81 | 55 | 9 | 34 | 0.61 | 0.61 | 60 |
| **Average** | **65.0** | **44.9** | **13.8** | **12.8** | **0.71** | **0.71** | **87.0** |

pared to MELT's 22.9%, despite both tools starting from the same validated examples. The fact that MELT can extract transformation rule from many of the synthetic examples provides additional evidence of their quality, and suggests the data generation pipeline may enhance other migration inference tools. However, MELT's rules are created in isolation. SPELL's agentic pipeline, by contrast, orchestrates these kinds of rules into coherent transformation strategies, effectively taking advantage of the expressive power of the POLYGLOTPIRANHA language. Figure 6.9 illustrates this difference: SPELL synthesizes interconnected rules with explicit scoping (per-function, per-file) that enable cascading transformations without overgeneralization; MELT produces disconnected global rules with generic placeholders, which can prove especially problematic on large codebases [184].

These results validate that (1) the generated synthetic examples provide a useful independent signal for migration inference, (2) the agentic approach to rule composition meaningfully improves over the current state-of-the-art, with intelligent orchestration of inferred rules importantly contributing to migration synthesis.

### 6.5.5 RQ4: Data Quality

**Methodology** We characterize the quality of the synthetically generated data across several dimensions:

1. **Idea implementation:** the number of the 100 generated ideas $I_j \in \mathcal{I}$ that yields at least one implementation $s_{j,k}$ that passes at least one generated test harness.
2. **Idea migration:** the number of implemented ideas that are migrated to the target library (how many ideas have at least one $s_i$ that is associated with a success $m_i$).
3. **API diversity:** number of distinct API methods used from both $\mathcal{S}$ and $\mathcal{T}$.[1] We compute this using the Jedi static analyzer [146] on validated triples $(s, t, m) \in \mathcal{M}$.

[1] Note that this is heuristic (due to `Python`'s dynamic typing), so it should be considered a lower bound.

4. **Test coverage:** via line-level coverage of the generated tests on both source and target implementations (coverage must be at least 0.6, per our definition of validity).
5. **Validated examples:** The total number of validated examples from all generated examples, post-filtering; this determines the training data available for script synthesis.

Together, these metrics characterizes the quality of the generated synthetic data, in terms of how well they cover or represent a migration task, and the degree to which they are adequately tested.

**Results.** Table 6.3 shows results across all data quality metrics. SPELL successfully implemented the majority of generated ideas, with an average of 65 out of 100 ideas yielding at least one working implementation. Success rates ranged from 24 (`jinja2 → mako`) to 96 (`json → orjson`). Upon manual inspection of `jinja2 → mako`, we found that most generated code could not run standalone, as a simple `python` script.

From the pool of ideas implemented at least once, SPELL successfully migrated an average of 44.9 ideas while maintaining functional equivalence per the generated tests. The outlier is `pandas → polars` with only 2 successful migrations despite 58 successful implementations. Manual inspection revealed that generated test harnesses relied on pandas-specific constructs (e.g., `DataFrames`) that were incompatible with `polars`.

Our analysis of API diversity using Jedi identified an average of 13.8 distinct source library APIs and 12.8 target library APIs per migration task. These likely correspond to the most frequently used APIs that the underlying LLM encountered during pre-training [183]. The `cryptography → pycryptodome` task showed the highest diversity (59 source, 25 target APIs), while simple tasks like json → orjson used fewer APIs (4 source, 3 target).

The generated tests achieved an average coverage of 71% for both source and target implementations, well above our 60% filtering threshold. Coverage ranged from 61% (`time → pendulum`) to 89%/93% (`argparse → click`). This suggests that tests meaningfully exercise implementation logic, while leaving room to improve; LLMs are known to struggle with unit test generation [185].

After applying all filtering criteria, SPELL produced an average of 87 validated migration triples per task (excluding `pandas → polars`). Overall, these examples are testable and diverse (covering an average of 13–14 distinct APIs per library), but also exhibit strong coverage (71% on average) on both source and migrated code. This supports their use as input for downstream synthesis.

## 6.6 Discussion and Limitations

**Quality and Representativeness of Synthetic Data.** Our pipeline relies on synthetic examples generated by a comparatively small model (`gpt4o-mini`) to keep the cost of data generation manageable; as the cost of more capable models decreases, we expect the generated examples to improve in quality.

Programs generated by LLMs are also biased toward the parts of each API that the model learned well.

Corner-case behaviours, error-handling paths, and rarely used configuration parameters are therefore under-represented. As larger models trained on more data become cheaper, we expect both the breadth of API usage and the semantic and syntactic correctness of the generated programs to improve. A complementary direction is to augment the idea–implementation prompts with retrieval over real code repositories.

**Transformation Expressiveness.** We represent our migration in the PolyglotPiranha language because of its expressive and concise rule-graph abstraction, as well as high performance. Nonetheless, PolyglotPiranha still has some limitations, in particular, it cannot yet rewrite code inside string literals, formatted templates (for example, `jinja2`), or dynamically generated constructs. This limitation was the main reason for the low success rate on the `jinja2→mako` task. Integrating multi-language parsing in PolyglotPiranha, or switching to a hybrid transformation engine that supports embedded DSLs could improve performance on this task.

Section 6.5 shows that the technique is promising in some cases and inadequate in others. One of the main limitations in real-world usage is the many variations of code found in the open source. The code transformation rules inferred capture concrete diff hunks faithfully but can be either too specific, failing to match slight syntactic variations, or too general, introducing false positives. Spell's agentic loop alleviates some of these issues by refining and testing rules, yet it remains fundamentally example-bound: if a usage pattern does not appear in any migration triple, the resulting script will not handle it. `Pycraft` [186]'s technique on rule generalization using LLMs could be integrated into our workflow to generate further equivalent script variations for a broader application.

**Tests.** A core pillar of Spell's validation process is the use of automatically generated tests to assess the correctness of both implementations and their migrated counterparts. While this enables large-scale validation without manual effort, it introduces some limitations. First, the quality of the tests is inherently tied to the generative model's ability to produce robust and meaningful test harnesses. Although we mitigate this by sampling multiple test files per implementation and applying filtering based on coverage, the tests may still be shallow or fail to exercise edge-case behavior. In particular, the tests are designed to confirm functional equivalence at a coarse granularity (e.g., returning the same output for a given input) but may miss subtle behavioral divergences, side effects, etc. Second, our approach assumes that passing the same test suite is a sufficient proxy for semantic equivalence. While this assumption is reasonable for many cases, it is not formally guaranteed. The test suites may lack completeness or specificity. Third, our coverage-based filtering helps ensure that the tests exercise a non-trivial portion of the implementation logic, but it does not guarantee semantic soundness. For instance, tests that exercise a large portion of the code but contain weak assertions (e.g., asserting only that no exceptions are raised) may still pass despite incorrect behavior. Other validation techniques (such as mutation testing [187]) could improve test quality and better capture subtle correctness issues.

**Data Leakage.** Unlike prior approaches, Spell does not depend on curated migration datasets or mined ver-

sion histories. Instead, it leverages the latent knowledge encoded in LLMs during pretraining. In practice, this means that SPELL's effectiveness depends on the extent to which the underlying model has been exposed to the source and target libraries, and potentially to migration examples between them during training. It is therefore possible that the model's outputs indirectly benefit from having "seen" real migrations in the wild, whether through public codebases, version histories, tutorials, or documentation. This raises a potential risk of data leakage [117] in evaluation: the model may be reproducing previously observed migration patterns rather than synthesizing them from first principles. However, unlike repository mining methods, SPELL does not require any explicit curation of migration examples, nor direct access to historical migration data. Verifying the degree of leakage would require inspecting the model's training corpus to identify overlapping migration examples, which is not feasible with current open-weight models.

## 6.7 Key Takeaways and Contributions

This chapter presented my work on SPELL, a technique for synthesizing library migration scripts using LLMs. The goal of this work is to show that, rather than relying on repository mining or manually curated examples, it is possible to systematically extract migration knowledge directly from LLMs and formalize it into reusable and testable migration scripts. The core idea is to treat LLMs as generators of aligned migration examples and to use test-based validation as a filter for correctness. This is possible because during pretraining LLMs implicitly learn how to map APIs across libraries by generalizing from a series of natural language constructs present in the code.

Beyond example generation, I also introduce an agentic strategy to generalize concrete examples into reusable migration scripts in the POLYGLOTPIRANHA language, which can then be automatically applied to other library clients. I show that SPELL can synthesize reusable and effective scripts for multiple `python` migrations. Finally, I discuss the limitations of this approach and outline avenues for future work in this area.

# 7

# Conclusion

**Contents**



## 7.1 Contributions

Refactoring is an important but laborious task. It ensures that code is better structured, easier to maintain, and more robust to future changes. However, most developers do not refactor as often as they should. In practice, companies incentivize feature delivery, not maintenance, and as a result, codebases grow in complexity and technical debt accumulates. To help developers refactor more efficiently, this thesis proposes a modern language and toolset for automating large-scale code transformations. On top of that, we present three approaches for automated API refactoring that do not rely on traditional sources of training data, which have limited general applicability over the years. Together, these contributions advance the state of the art in three ways. First, they show that it is possible to automate complex migrations without relying on mined client examples. Second, the transformations are expressed in a concise and human-readable format, allow-

ing developers to review and modify them. Third, our tools are scalable and fast, and can be applied to real codebases in realistic settings.

## 7.2 Open Challenges and Future Work

Ultimately, the impact of automated code migration will be measured by its adoption in practice. Although research prototypes are abundant, industry adoption has historically been limited [66]. This is now beginning to change as companies start to deploy LLM-based migration tools and report some success (although in very limited domains, such as migrating int data types [119]). To accelerate this trend, multiple open challenges must be addressed.

### 7.2.1 Correctness and Trust

Despite the progress made in automated software engineering technology, significant technical challenges remain before tools can reach their full potential. In case of refactoring, one major technical hurdle is ensuring correctness and behavior preservation of transformed code. Developers will not adopt tools they do not trust, especially for large-scale changes [58]. Thus, automated refactoring tooling must convincingly demonstrate that it has not broken the original code. Several approaches can contribute to increasing trust, including rigorous validation of correctness, explainability of transformations, and mechanisms that help developers assess the confidence of the tool's suggestions.

Most automated refactoring techniques use test-driven validation to verify correctness, which provides reasonable confidence that the behavior is unchanged. However, for this to work, test suites must be exhaustive; otherwise gaps in test coverage can allow subtle bugs to slip through. Future migration tools can explore deep integration with testing infrastructure, that is, instead of treating tests only as a final check, the tools could use tests to guide the migration itself. For instance, the tool might generate targeted inputs for the test suite to increase coverage of the code regions being refactored and prevent regressions. Another idea is differential testing: run both the old and new versions of the software on the same inputs (perhaps using recorded production traffic, if available) to ensure they behave equivalently. This could catch differences that the original test suite might miss.

Beyond traditional testing, static analysis can provide another layer of assurance. Although full formal verification of arbitrary code transformations is unrealistic, targeted verification of certain properties could be useful. For example, in security-sensitive migrations (like changing cryptography libraries), the tooling could verify that the new code respects the same security invariants (no weaker cipher, proper error handling, etc). Refactoring tools could incorporate model check or symbolic reasoning for specific known-critical properties [188].

Explainability is another crucial ingredient for trust. Developers are more likely to trust a change if they

understand why it was made and how [189]. Automated migration systems of the future should thus explain their actions and make sure changes can be reviewed in digestible chunks. This could mean linking each change to a rationale, such as "*Replaced this function call because the older API is deprecated and the documentation recommends using the new API*". By providing these explanations in-line or in commit messages, the tool can help human reviewers quickly gain confidence that the change is legitimate. Ensuring these explanations are accurate (and not hallucinated) will require careful design, possibly having the explanations validated against the same documentation sources.

In the near future, as researchers and practitioners pursue partial automation of migration processes, it will be useful to define the scope of tasks and break them down into chunks that can be automated. This means the tool should defer to human judgment when the situation is ambiguous or when policy decisions are involved (for example, choosing whether a migration is safe if behavior is not always preserved). Conversely, the tool should take over mundane tasks and only bother humans for review or approval. Achieving this balance requires UI and workflow design that gives developers control, as we further discuss in Section 7.2.4. One approach is to provide confidence indicators: the tool can indicate its confidence level in each change. Another approach is a trust calibration framework, as suggested by recent work [190], which categorizes the tool's outputs (e.g., correct suggestions, incorrect suggestions, partial edits) and helps train users to understand how to handle each category. For example, if a suggestion is labeled as "low confidence," a developer might double-check it more thoroughly.

### 7.2.2 Stateful and Complex Transformations

Automated code migration is poised to benefit from several emerging research trends in artificial intelligence for software engineering (AI4SE). In particular, LLMs have demonstrated strong automation capabilities and have already been applied to assist in code transformation tasks. There are clear opportunities to extend and better leverage these models to build tooling for automating migrations. This thesis shows that even without curated migration datasets, it is possible to extract migration knowledge from models, formalize it into reusable refactoring scripts, and verify that knowledge before applying it. Future systems can build on this insight, making use of the implicit knowledge encoded in LLMs.

Emerging research is also exploring agent-based approaches that treat code migration as a search or game, where an AI agent plans and performs iterative transformations to achieve the end goal. While most current work focuses on leveraging generalist models for automated migration, there is potential to extend these ideas into more general reinforcement learning or planning frameworks specifically designed for this task. An agent could, for example, learn a policy for applying a sequence of small refactorings to achieve a larger migration goal, guided by rewards such as passing all tests. This would shift the problem from simple prompting approaches to more principled, stepwise optimization processes capable of handling complex migrations through decomposition. This is particularly relevant because, although agentic approaches with

preprocessing steps have potential, practitioners have noted that with current technology, planning might even be harmful in some cases [191]. Nevertheless, this area remains largely unexplored and presents a rich research opportunity at the intersection of software engineering and AI planning.

On the infrastructure and language support side, handling complex, stateful transformations remains a significant challenge. Many API migrations are not simple one-to-one method renames: they may involve reordering calls, managing additional state, or updating types and invariants. Existing rule-based systems struggle with this, as they tend to focus on localized code changes. Enhancing tools to deal with non-local or context-dependent migrations is therefore a priority. Combining structured transformations with more ad hoc edits using LLMs may be the most practical path forward. For instance, migrating a library might require adding initialization code in one part of the application and cleanup code elsewhere, or splitting a single call into multiple calls. Future migration DSLs and engines need to represent such multi-step transformations more cleanly. One promising direction is to incorporate temporal logic or graph transformations to capture sequences of code edits that must occur together [86].

### 7.2.3 Performance and Scalability

Performance and scalability are critical for automated refactoring tools. As codebases grow to millions of lines of code and span hundreds of services or repositories, migration tools must be able to operate efficiently at this scale. The techniques proposed in this thesis were designed with scalability in mind and have demonstrated that performant, large-scale transformations are possible. Future work should continue to optimize and leverage production ready tooling to support real-world migration scenarios.

A related challenge is multi-language support. Large systems often consist of multiple programming languages and frameworks, and real-world migrations may need to modify code across languages. For example, updating both backend Java code and frontend TypeScript, or synchronizing code and configuration file changes. While this thesis introduced mechanisms to express multi-language transformations, there is room for further development. Future tools will need to handle heterogeneous codebases, either by unifying syntax and semantics in a common representation or by leveraging LLMs to fill gaps in language-agnostic tooling.

Finally, applying changes at scale requires orchestration beyond generating the patch itself. This includes managing rollouts, coordinating changes across repositories, running CI pipelines, and handling code reviews and merges. Future migration frameworks could integrate orchestration layers that automate these tasks, for example by creating branches across multiple repositories, applying patches, running tests, and submitting pull requests or merging changes if all checks pass.

### 7.2.4 Usability and Workflow Integration

Usability is a key factor in the adoption of automated refactoring tools. Prior work and our own findings suggest that poor usability is one of the main reasons developers avoid or abandon such tools. A common issue is that tool-generated edits can be hard to review, especially when they come as large, monolithic patches that touch hundreds or thousands of files. These changes can overwhelm reviewers, disrupt version control history, and make code reviews unmanageable.

Finally, there is a growing need to manage the volume of auto-generated changes. Although modern tools can generate more refactoring suggestions than ever, developers still need to review them, and generating too much code might be counterproductive. Future work should explore ways to filter, prioritize, and present only the most relevant changes to human reviewers. Finding better ways to present information to humans will be critical for making automated migration tooling usable in practice.

## 7.3 Final Remarks

While automated refactoring has seen decades of research, its industry adoption has remained limited [66]. This is slowly starting to change. We are now seeing major companies integrate refactoring automation into their workflows, in particular through the use of LLMs. Commercial tools like Amazon Q [118], Google's LLM-integrated infrastructure [119], and startups like Cursor [192] are using these models to handle library upgrades and API migrations at scale.

This thesis provides further evidence that the traditional reliance on mined migration examples is not strictly necessary. Instead, drawing from documentation, internal library evolution, and the implicit knowledge of large models, we can build systems that generalize more broadly for refactoring. These findings align with what is already being observed in practice. What this work offers is a principled way to formalize and validate that process, with a stronger emphasis on testability, script synthesis, and correctness.

With emerging capabilities of LLM's and agentic approaches, we expect interest in the automated migration space to continue to grow. However, automated refactoring technology is not yet mature enough to be universally adopted. Limited support for complex stateful transformations, as well as LLM hallucinations and integration with developer workflows remain open challenges. As more robust combinations of symbolic and generative tools emerge, we expect the next few years to be a particularly exciting time for automated refactoring, both in research and in practice.

# Bibliography


[1] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," in *ECOOP*, ser. Lecture Notes in Computer Science, vol. 5653, Springer, 2009, pp. 27–51.

[2] 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.

[3] C. R. de Souza and D. F. Redmiles, "On the roles of apis in the coordination of collaborative software development," *Computer Supported Cooperative Work (CSCW)*, vol. 18, no. 5-6, p. 445, 2009.

[4] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 9, pp. 1264–1282, 2013.

[5] A. Ketkar, D. Ramos, L. Clapp, R. Barik, and M. K. Ramanathan, "A Lightweight Polyglot Code Transformation Language," in *Proceedings of the ACM on Programming Languages*, ser. PLDI '24, ACM, 2024, DOI: 10.1145/3656429.

[6] V. Rajlich, "Software evolution and maintenance," in *Future of Software Engineering Proceedings*, 2014, pp. 133–144.

[7] 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.

[8] 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.

[9] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 2, pp. 126–139, 2004.

[10] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of Github contributors," in *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 858–870.

[11] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, E. D. Nitto, M. Harman, and P. Heymans, Eds., ACM, 2015, pp. 50–60.

[12] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 1, pp. 1–12, 2001.

[13] H. Krasner, *The cost of poor code quality in the us: A 2022 report*, Accessed: 28-09-2023, CISQ, 2022, [Online]. Available: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf.

[14] 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)*, 2012, pp. 1–11.

[15] *Pytorch*, https://pytorch.org/, Accessed: 2023-11-20, 2023.

[16] *Api documentation : Tensorflow core v2.2.0*, https://www.tensorflow.org/api_docs/index.html, Dec. 2023.

[17] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar apis: An exploratory study," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 266–276.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[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] GitHub, *Github: Where the world builds software*, https://github.com, Accessed: 2023-11-22, 2023.

[25] 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.

[26] 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.

[27] 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.

[28] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deepam: Migrate apis with multi-modal sequence to sequence learning,” *arXiv preprint arXiv:1704.07734*, 2017.

[29] H. Zhong, L. Zhang, T. Xie, and H. Mei, “Inferring resource specifications from natural language API documentation,” in *Proc. International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2009, pp. 307–318.

[30] G. Uddin and M. P. Robillard, “How API documentation fails,” *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[31] R. B. Watson, M. Stamnes, J. Jeannot-Schroeder, and J. H. Spyridakis, “API documentation and software community values: A survey of open-source API documentation,” in *Proceedings of the 31st ACM international conference on Design of communication, Greenville, NC, USA, September 30 - October 1, 2013*, M. J. Albers and K. Gossett, Eds., ACM, 2013, pp. 165–174, [Online]. Available: https://doi.org/10.1145/2507065.2507076.

[32] T. Lethbridge, J. Singer, and A. Forward, “How software engineers use documentation: The state of the practice,” *IEEE Softw.*, vol. 20, no. 6, pp. 35–39, 2003, [Online]. Available: https://doi.org/10.1109/MS.2003.1241364.

[33] W. Jin, D. Zhong, Z. Ding, M. Fan, and T. Liu, “Where to start: Studying type annotation practices in python,” in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, IEEE, 2021, pp. 529–541, [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678947.

[34] S. M. Sohan, C. Anslow, and F. Maurer, “A case study of web API evolution,” in *2015 IEEE World Congress on Services, SERVICES 2015, New York City, NY, USA, June 27 - July 2, 2015*, L. Zhang and R. Bahsoon, Eds., IEEE Computer Society, 2015, pp. 245–252, [Online]. Available: https://doi.org/10.1109/SERVICES.2015.43.

[35] 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.

[36] 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.

[37] M. D. Ernst, “Natural language is a programming language: Applying natural language processing to software development,” in *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., ser. LIPIcs, vol. 71, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 4:1–4:14, [Online]. Available: https://doi.org/10.4230/LIPIcs.SNAPL.2017.4.

[38] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, and Y. L. Traon, “Got issues? who cares about it? A large scale investigation of issue trackers from github,” in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, IEEE Computer Society, 2013, pp. 188–197, [Online]. Available: https://doi.org/10.1109/ISSRE.2013.6698918.

[39] OpenAI, *GPT-4*, https://openai.com/research/gpt-4/, Accessed: May 2, 2023.

[40] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, “Unsupervised translation of programming languages,” *arXiv preprint arXiv:2006.03511*, 2020.

[41] S. Gulwani, O. Polozov, and R. Singh, “Program synthesis,” *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017, [Online]. Available: https://doi.org/10.1561/2500000010.

[42] K. L. Gwet, *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*. Advanced Analytics, LLC, 2014.

[43] 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, 55:1–55:42, 2021.

[44] A. Ni *et al.*, “SOAR: A synthesis approach for data science API refactoring,” in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 112–124.

[45] D. Ramos, H. Mitchell, I. Lynce, V. M. Manquinho, R. Martins, and C. L. Goues, “MELT: mining effective lightweight transformations from pull requests,” in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, IEEE, 2023, pp. 1516–1528, [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00117.

[46] M. Reddy, *API Design for C++*. Elsevier, 2011.

[47] J. Bloch, *A brief history of the api*, InfoQ, Presentation at QCon San Francisco, 2018, [Online]. Available: https://www.infoq.com/presentations/history-api/.

[48] J. Ofoeda, R. Boateng, and J. Effah, "Application programming interface (API) research: A review of the past to inform the future," *Int. J. Enterp. Inf. Syst.*, vol. 15, no. 3, pp. 76–95, 2019, [Online]. Available: https://doi.org/10.4018/IJEIS.2019070105.

[49] J. J. Bloch, "How to design a good API and why it matters," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, P. L. Tarr and W. R. Cook, Eds., ACM, 2006, pp. 506–507, [Online]. Available: https://doi.org/10.1145/1176617.1176622.

[50] K. Cwalina, J. Barton, and B. Abrams, *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Addison-Wesley Professional, 2020.

[51] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "API designers in the field: Design practices and challenges for creating usable apis," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2018, Lisbon, Portugal, October 1-4, 2018*, J. Cunha, J. P. Fernandes, C. Kelleher, G. Engels, and J. Mendes, Eds., IEEE Computer Society, 2018, pp. 249–258, [Online]. Available: https://doi.org/10.1109/VLHCC.2018.8506523.

[52] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[53] B. Ellis, J. Stylos, and B. A. Myers, "The factory pattern in API design: A usability evaluation," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, IEEE Computer Society, 2007, pp. 302–312, [Online]. Available: https://doi.org/10.1109/ICSE.2007.85.

[54] V. Rajlich, "Software evolution and maintenance," in *Future of Software Engineering Proceedings*, 2014, pp. 133–144.

[55] H. C. Benestad, B. Anda, and E. Arisholm, "Understanding cost drivers of software evolution: A quantitative and qualitative investigation of change effort in two evolving software systems," *Empirical Software Engineering*, vol. 15, pp. 166–203, 2010.

[56] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: An investigation of how developers spend their time," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, A. D. Lucia, C. Bird, and R. Oliveto, Eds., IEEE Computer Society, 2015, pp. 25–35, [Online]. Available: https://doi.org/10.1109/ICPC.2015.12.

[57] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[58] M. Kim, T. Zimmermann, and N. Nagappan, “An empirical study of refactoringchallenges and benefits at Microsoft,” *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 7, pp. 633–649, 2014.

[59] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, “Mining API mapping for language migration,” in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds., ACM, 2010, pp. 195–204.

[60] D. Dig and R. E. Johnson, “How do apis evolve? A story of refactoring,” *J. Softw. Maintenance Res. Pract.*, vol. 18, no. 2, pp. 83–107, 2006, [Online]. Available: https://doi.org/10.1002/smr.328.

[61] C. Vassallo, F. Palomba, and H. C. Gall, “Continuous refactoring in CI: A preliminary study on the perceived advantages and barriers,” in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, IEEE Computer Society, 2018, pp. 564–568, [Online]. Available: https://doi.org/10.1109/ICSME.2018.00068.

[62] T. Sharma, G. Suryanarayana, and G. Samarthyam, “Challenges to and solutions for refactoring adoption: An industrial perspective,” *IEEE Software*, vol. 32, no. 6, pp. 44–51, 2015.

[63] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[64] M. Kim, D. Cai, and S. Kim, “An empirical investigation into the role of api-level refactorings during software evolution,” in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 151–160.

[65] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, “When does a refactoring induce bugs? an empirical study,” in *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*, IEEE Computer Society, 2012, pp. 104–113, [Online]. Available: https://doi.org/10.1109/SCAM.2012.20.

[66] J. Ivers, R. L. Nord, I. Ozkaya, C. Seifried, C. S. Timperley, and M. Kessentini, “Industry’s cry for tools that support large-scale refactoring,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 163–164.

[67] E. Tempero, T. Gorschek, and L. Angelis, “Barriers to refactoring,” *Communications of the ACM*, vol. 60, no. 10, pp. 54–61, 2017.

[68] E. Murphy-Hill and A. P. Black, “Refactoring tools: Fitness for purpose,” *IEEE software*, vol. 25, no. 5, pp. 38–44, 2008.

[69] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, “Use, disuse, and misuse of automated refactorings,” in *2012 34th international conference on software engineering (icse)*, IEEE, 2012, pp. 233–243.

[70] J. Lawall and G. Muller, “Coccinelle: 10 years of automated evolution in the linux kernel,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 601–614.

[71] L. Wasserman, “Scalable, example-based refactorings with refaster,” in *Workshop on Refactoring Tools*, 2013, [Online]. Available: http://dx.doi.org/10.1145/2541348.2541355.

[72] H. J. Kang, T. Ferdian, J. Lawall, G. Muller, L. Jiang, and D. Lo, “Semantic patches for java program transformation (artifact),” 2019.

[73] I. Uber Technologies, *Go-patch: Structured code diffs and refactors*, https://github.com/uber-go/gopatch, GitHub repository, 2023, [Online]. Available: https://github.com/uber-go/gopatch.

[74] Clang. “Clang libtooling.” Accessed: 2024-03-06. (2024).

[75] aftandilian2012buildingOpen-source community, *Ast-grep: Write code to match code*, Accessed: 2023-09-22, 2023, [Online]. Available: https://ast-grep.github.io.

[76] L. C. L. Kats and E. Visser, “The spoofax language workbench: Rules for declarative specification of languages and ides,” in *Proc. ACM SIGPLAN Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds., Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 444–463, [Online]. Available: https://doi.org/10.1145/1869459.1869497.

[77] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, “Stratego/xt 0.17. a language and toolset for program transformation,” *Science of computer programming*, vol. 72, no. 1-2, pp. 52–70, 2008.

[78] P. Klint, T. van der Storm, and J. J. Vinju, “RASCAL: A domain specific language for source code analysis and manipulation,” Edmonton, Alberta, Canada: IEEE Computer Society, 2009, pp. 168–177, [Online]. Available: https://doi.org/10.1109/SCAM.2009.28.

[79] DMS Toolkit. “DMS Toolkit.” Available: http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html, Semantic Designs. (2024), (visited on 03/06/2024).

[80] J. R. Cordy, “Txl-a language for programming language tools and applications,” *Electronic notes in theoretical computer science*, vol. 110, pp. 3–31, 2004.

[81] J. Koppel, V. Premtoon, and A. Solar-Lezama, “One tool, many languages: Language-parametric transformation with incremental parametric syntax,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[82] GitHub. “Codeql: The libraries and queries that power security researchers around the world, as well as code scanning in github advanced security,” GitHub, Inc. (2024), [Online]. Available: https://codeql.github.com.

[83] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 1037–1039, ISBN: 978-1-4503-0445-0, [Online]. Available: http://doi.acm.org/10.1145/1985793.1985989.

[84] Y. Tang, R. Khatchadourian, M. Bagherzadeh, and S. Ahmed, "Towards safe refactoring for intelligent parallelization of Java 8 streams," in *International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '18, ACM/IEEE, Gothenburg, Sweden: ACM, May 2018, pp. 206–207, ISBN: 978-1-4503-5663-3, [Online]. Available: http://academicworks.cuny.edu/hc_pubs/355.

[85] L. Franklin, A. Gyori, J. Lahoda, and D. Dig, "Lambdaficator: From imperative to functional programming through automated refactoring," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1287–1290, DOI: 10.1109/ICSE.2013.6606699.

[86] A. Ketkar, A. Mesbah, D. Mazinanian, D. Dig, and E. Aftandilian, "Type migration in ultra-large-scale codebases," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, J. M. Atlee, T. Bultan, and J. Whittle, Eds., Montreal, QC, Canada: IEEE / ACM, 2019, pp. 1142–1153, [Online]. Available: https://doi.org/10.1109/ICSE.2019.00117.

[87] R. Ossendrijver, S. Schroevers, and C. Grelck, "Towards automated library migrations with error prone and refaster," ser. SAC '22, Virtual Event: Association for Computing Machinery, 2022, pp. 1598–1606, ISBN: 9781450387132, [Online]. Available: https://doi.org/10.1145/3477314.3507153.

[88] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05, San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 265–279, ISBN: 1595930310, [Online]. Available: https://doi.org/10.1145/1094811.1094832.

[89] Online. "Open Rewrite Recipes." Available: https://docs.openrewrite.org/concepts-explanations/recipes. (2024), (visited on 03/06/2024).

[90] M. K. Ramanathan, L. Clapp, R. Barik, and M. Sridharan, "Piranha: Reducing feature flag debt at uber," in *Proc. International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 221–230.

[91] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," Riva del Garda, Italy: IEEE Computer Society, 2012, pp. 14–23, [Online]. Available: https://doi.org/10.1109/SCAM.2012.28.

[92] Moderne. "OpenRewrite: Semantic code search and transformation tool." Accessed: 2024-03-06, Moderne Inc. (2024), [Online]. Available: https://docs.openrewrite.org.

[93] GitHub, *Tree-sitter*, Accessed: 2023-09-24, 2023, [Online]. Available: https://tree-sitter.github.io/tree-sitter/.

[94] R. Rolim *et al.*, "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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, "Exploring API method parameter recommendations," in *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME)*, R. Koschke, J. Krinke, and M. P. Robillard, Eds., 2015, pp. 271–280.

[100] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *Proc. International Conference on Automated Software Engineering (ASE)*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds., ACM, 2014, pp. 457–468.

[101] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of API usages," in *Proc. ACM IEEE International Conference on Software Engineering (ICSE)*, S. Uchitel, A. Orso, and M. P. Robillard, Eds., 2017, pp. 47–50.

[102] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 502–511, ISBN: 9781467330763.

[103] A. Miltner *et al.*, "On the fly synthesis of edit suggestions," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019, [Online]. Available: https://doi.org/10.1145/3360569.

[104] Y. Zhang *et al.*, "Overwatch: Learning patterns in code edit sequences," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 395–423, 2022, [Online]. Available: https://doi.org/10.1145/3563302.

[105] 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.

[106] Z. Feng *et al.*, “Codebert: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, T. Cohn, Y. He, and Y. Liu, Eds., ser. Findings of ACL, vol. EMNLP 2020, Association for Computational Linguistics, 2020, pp. 1536–1547, [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[107] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, “Unified pre-training for program understanding and generation,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, K. Toutanova *et al.*, Eds., Association for Computational Linguistics, 2021, pp. 2655–2668, [Online]. Available: https://doi.org/10.18653/v1/2021.naacl-main.211.

[108] Y. Wang, H. Le, A. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, “Codet5+: Open code large language models for code understanding and generation,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, H. Bouamor, J. Pino, and K. Bali, Eds., Association for Computational Linguistics, 2023, pp. 1069–1088, [Online]. Available: https://doi.org/10.18653/v1/2023.emnlp-main.68.

[109] E. Nijkamp *et al.*, “Codegen: An open large language model for code with multi-turn program synthesis,” in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*, OpenReview.net, 2023, [Online]. Available: https://openreview.net/forum?id=iaYcJKpY2B%5C_.

[110] Y. Li *et al.*, “Competition-level code generation with alphacode,” *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[111] D. Pomian *et al.*, “Next-generation refactoring: Combining LLM insights and IDE capabilities for extract method,” in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*, IEEE, 2024, pp. 275–287, [Online]. Available: https://doi.org/10.1109/ICSME58944.2024.00034.

[112] D. Pomian *et al.*, “Em-assist: Safe automated extractmethod refactoring with llms,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, M. d’Amorim, Ed., ACM, 2024, pp. 582–586, [Online]. Available: https://doi.org/10.1145/3663529.3663803.

[113] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, “An empirical study on the potential of llms in automated software refactoring,” *arXiv preprint arXiv:2411.04444*, 2024.

[114] C. Cummins, V. Seeker, J. Armengol-EstapÃŠ, A. H. Markosyan, G. Synnaeve, and H. Leather, “Don’t transform the code, code the transforms: Towards precise code rewriting using llms,” *arXiv preprint arXiv:2410.08806*, 2024.

[115] J. D. Rocco, P. T. Nguyen, C. D. Sipio, R. Rubei, D. D. Ruscio, and M. D. Penta, "Deepmig: A transformer-based approach to support coupled library and code migrations," *Inf. Softw. Technol.*, vol. 177, p. 107 588, 2025, [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107588.

[116] M. M. Islam, A. K. Jha, M. Mahmoud, I. Akhmetov, and S. Nadi, "Using llms for library migration," *arXiv preprint arXiv:2504.13272*, 2025.

[117] D. Ramos, C. Mamede, K. Jain, P. Canelas, C. Gamboa, and C. L. Goues, "Are large language models memorizing bug benchmarks?" *arXiv preprint arXiv:2411.13323*, 2024.

[118] A. W. Services, *Amazon q developer: Transform*, https://aws.amazon.com/q/developer/transform/, Accessed: 2025-03-27, 2025.

[119] C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, "Migrating code at scale with llms at google," *arXiv preprint arXiv:2504.09691*, 2025.

[120] *Pytorch conv2d api documentation*, https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html, Dec. 2023.

[121] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[122] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP*, ACL, 2014, pp. 1532–1543.

[123] M. F. Porter, *Snowball: A language for stemming algorithms*, 2001.

[124] C. S. Perone, R. Silveira, and T. S. Paula, "Evaluation of sentence embeddings in downstream and linguistic probing tasks," *arXiv preprint arXiv:1806.06259*, 2018.

[125] S. Arora, Y. Liang, and T. Ma, "A simple but tough-to-beat baseline for sentence embeddings," in *ICLR (Poster)*, OpenReview.net, 2017.

[126] *Common crawl*, https://commoncrawl.org/, Dec. 2023.

[127] A. Solar-Lezama, "The sketching approach to program synthesis," in *APLAS*, ser. Lecture Notes in Computer Science, vol. 5904, Springer, 2009, pp. 4–13.

[128] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho, "Encodings for enumeration-based program synthesis," in *CP*, ser. Lecture Notes in Computer Science, vol. 11802, Springer, 2019, pp. 583–599.

[129] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013, pp. 3111–3119.

[130] M. A. Hearst, "Automatic acquisition of hyponyms from large text corpora," in *COLING*, 1992, pp. 539–545.

[131] *Tensorflow tutorial*, https://www.tensorflow.org/tutorials, Dec. 2023.

[132] *Tensorflow applications*, https://www.tensorflow.org/apidocs/python/tf/keras/applications, Dec. 2023.

[133] *Tensorflow models*, https://github.com/tensorflow/models, Dec. 2023.

[134] *Kaggle*, https://www.kaggle.com, Dec. 2023.

[135] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *PLDI*, ACM, 2017, pp. 422–436.

[136] *Scrapy: A fast and powerful scraping and web crawling framework*, https://scrapy.org/, Dec. 2023.

[137] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340.

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

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

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

[141] *Scipy: Open source scientific tools for python*, https://www.scipy.org/, Accessed: May 2, 2023, 2023.

[142] *Deprecate signal.spline in favor of signal*, https://github.com/scipy/scipy/pull/14419, Accessed: May 2, 2023.

[143] 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.

[144] *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.

[145] 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.

[146] D. Halter, *Jedi: An autocompletion tool for python*, https://jedi.readthedocs.io/en/latest/, Accessed: May 2, 2023.

[147] T. S. BV, *Tiobe index*, https://www.tiobe.com/tiobe-index/, Accessed on May 3, 2023, 2023.

[148] R. van Tonder, *Comby with types*, https://comby.dev/blog/2022/08/31/comby-with-types, Accessed on 3 May 2023, Aug. 2022.

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

[150] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960, ISSN: 0013-1644.

[151] 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.

[152] *Dep: Deprecate rollaxis*, GitHub Pull Request Numpy #9475, Accessed on May 4, 2023 https://github.com/numpy/numpy/pull/9475.

[153] *DEPR: Pd.read_table*, GitHub Pull Request Pandas #21954, Accessed on May 4, 2023 https://github.com/pandas-dev/pandas/pull/21954.

[154] *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.

[155] *Sourcegraph*, https://sourcegraph.com/, Accessed on May 5, 2023.

[156] Claude, *100k context windows*, Anthropic, Accessed on July 24, 2023, https://www.anthropic.com/index/100k-context-windows, 2023.

[157] T. Winters, T. Manshreck, and H. Wright, "Large-scale changes," in *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media, 2020, ch. 22.

[158] J. Kim, D. Batory, and D. Dig, "Scripting parametric refactorings in java to retrofit design patterns," in *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2015, pp. 211–220.

[159] M. Inc., *Fastmod*, Accessed: 2023-09-22, 2023, [Online]. Available: https://github.com/facebookincubator/fastmod.

[160] A. Bosch and Contributors, *Detekt: A static code analyzer for Kotlin*, Accessed: 2023-09-21, 2023, [Online]. Available: https://detekt.dev.

[161] J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2006, ISBN: 978-0-596-52812-6.

[162] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley series in computer science / World student series edition). Boston, MA, USA: Addison-Wesley, 1986, ISBN: 0-201-10088-6, [Online]. Available: https://www.worldcat.org/oclc/12285707.

[163] M. Brunsfeld, *Tree-sitter: A new parsing system for programming tools*, Strange Loop Conference, Available online: https://thestrangeloop.com/2018/tree-sitter---a-new-parsing-system-for-programming-tools.html, St. Louis, MO: Strange Loop, Sep. 2018.

[164] N. G. Fruja, "The correctness of the definite assignment analysis in c#," *J. Object Technol.*, vol. 3, no. 9, pp. 29–52, 2004, [Online]. Available: https://doi.org/10.5381/jot.2004.3.9.a2.

[165] M. T. Rahman, L. Querel, P. C. Rigby, and B. Adams, "Feature toggles: Practitioner practices and a case study," in *Proc. ACM IEEE International Conference on Mining Software Repositories (MSR)*, M. Kim, R. Robbes, and C. Bird, Eds., Austin, TX, USA: ACM, 2016, pp. 201–211, [Online]. Available: https://doi.org/10.1145/2901739.2901745.

[166] M. T. Rahman, P. C. Rigby, and E. Shihab, "The modular and feature toggle architectures of google chrome," *Springer Empirical Software Engineering (ESE)*, vol. 24, no. 2, pp. 826–853, 2019, [Online]. Available: https://doi.org/10.1007/s10664-018-9639-0.

[167] Apple. "Swiftsyntax documentation." Accessed: 2024-03-06, Apple Inc. (2024), [Online]. Available: https://swiftpackageindex.com/apple/swift-syntax/509.0.0/documentation/swiftsyntax.

[168] Meta, "Build faster with buck2: Our open source build system," Apr. 2024, Accessed: 2023-11-14, [Online]. Available: https://engineering.fb.com/2023/04/06/open-source/buck2-open-source-large-scale-build-system/.

[169] Online. "ErrorProne ComplexBooleanConstant." Available: https://errorprone.info/bugpattern/UnusedMethod. (2024), (visited on 03/06/2024).

[170] Online. "ErrorProne ComplexBooleanConstant." Available: https://errorprone.info/bugpattern/ComplexBooleanConstant. (2024), (visited on 03/06/2024).

[171] S. Banerjee, L. Clapp, and M. Sridharan, "Nullaway: Practical type-based null safety for java," in *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds., Tallinn, Estonia: ACM, 2019, pp. 740–750, [Online]. Available: https://doi.org/10.1145/3338906.3338919.

[172] Online. "Fix cwe-338 with securerandom." Available: https://docs.openrewrite.org/recipes/java/security/fixcwe338. (2024), (visited on 03/06/2024).

[173] Online. "Loggers should be named for their enclosing classes." Available: https://docs.openrewrite.org/recipes/java/logging/slf4j/loggersnamedforenclosingclass. (2024), (visited on 03/06/2024).

[174] Online. "Use secure temporary file creation." Available: https://docs.openrewrite.org/recipes/java/security/securetempfilecreation. (2024), (visited on 03/06/2024).

[175] 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.

[176] G. Team, *GritQL: Code transformation query language*, https://grit.io/gritql/, Accessed: 2025-03-27, 2025.

[177] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, "Large language models and simple, stupid bugs," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, IEEE, 2023, pp. 563–575.

[178] Y. Wang *et al.*, "Self-instruct: Aligning language models with self-generated instructions," *arXiv preprint arXiv:2212.10560*, 2022.

[179] U. T. Inc., *Piranha: A tool for refactoring code related to feature flag APIs*, Accessed: 2025-04-10, 2025, [Online]. Available: https://github.com/uber/piranha.

[180] OpenAI, *OpenAI Python API Library*, https://pypi.org/project/openai/, Accessed: April 22, 2025.

[181] C. Snell, J. Lee, K. Xu, and A. Kumar, "Scaling llm test-time compute optimally can be more effective than scaling model parameters," *arXiv preprint arXiv:2408.03314*, 2024.

[182] OpenAI. "Gpt-4.1: Introducing gpt-4.1 in the api." (Apr. 14, 2025), [Online]. Available: https://openai.com/index/gpt-4-1/ (visited on 06/08/2025).

[183] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[184] A. Ketkar, D. Ramos, L. Clapp, R. Barik, and M. K. Ramanathan, "A lightweight polyglot code transformation language," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, pp. 1288–1312, 2024, [Online]. Available: https://doi.org/10.1145/3656429.

[185] N. Rao, K. Jain, U. Alon, C. Le Goues, and V. J. Hellendoorn, "Cat-lm training language models on aligned code and tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 409–420.

[186] M. Dilhara, A. Bellur, T. Bryksin, and D. Dig, "Unprecedented code change automation: The fusion of llms and transformation by example," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 631–653, 2024.

[187] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: An analysis and survey," in *Advances in computers*, vol. 112, Elsevier, 2019, pp. 275–378.

[188] C. Foster *et al.*, "Mutation-guided llm-based test generation at meta," *arXiv preprint arXiv:2501.12862*, 2025.

[189] L. Kästner, M. Langer, V. Lazar, A. Schomäcker, T. Speith, and S. Sterz, "On the relation of trust and explainability: Why to engineer for trustworthiness," in *2021 IEEE 29th international requirements engineering conference workshops (REW)*, IEEE, 2021, pp. 169–175.

[190] B. Omidvar Tehrani and A. Anubhai, "Evaluating human-ai partnership for llm-based code migration," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–8.

[191] S. Nikolov *et al.*, "How is google using ai for internal code migrations?" *arXiv preprint arXiv:2501.06972*, 2025.

[192] Anysphere Inc., *Cursor: An ai-powered code editor*, version 1.0, AI-powered integrated development environment; fork of Visual Studio Code, Anysphere Inc., 2023, [Online]. Available: https://www.cursor.com/.