# PromiseAwait: A Dataset of JavaScript Migrations from Promises to Async/Await

Rafael Araujo Magesty
rafaelmagesty@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

João Eduardo Montandon
joao@dcc.ufmg.br
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

## Abstract

The constant evolution of the JavaScript language brings modern language features that simplify software development, notably the async/await syntax for asynchronous programming. While these advancements improve code readability and maintainability, migrating legacy codebases to leverage new constructs remains a complex and error-prone process. To support research on automated language migration, we present PromiseAwait, a comprehensive dataset of real-world JavaScript migrations from Promises to async/await syntax. At its current version, the dataset comprises 4,221 migration instances extracted from 1,530 commits across 408 popular open-source projects, identified using a heuristic based on regular expressions. Each instance includes detailed metadata and source code before and after the transition. We believe PromiseAwait will allow researchers to evaluate large language models for language migration tasks, providing a baseline for future studies.

**ACM Reference Format:**
Rafael Araujo Magesty and João Eduardo Montandon. 2026. PromiseAwait: A Dataset of JavaScript Migrations from Promises to Async/Await. In *2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering (FORGE '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793655.3793714

## 1 Introduction

JavaScript has become one of the most popular programming languages in the world, being frequently recognized as the de-facto language for many software developers [17, 22, 23]. Originally designed for client-side web development [7, 13], JavaScript has evolved significantly over the years, expanding its adoption to other application domains. Nowadays, software developers use JavaScript to build both front-end and back-end applications, mobile apps, and desktop software [5, 9, 13, 17, 20].

To support this diverse range of applications, JavaScript has been continuously updated with new features and syntax improvements [8]. Just a decade ago, the language started supporting class-based syntax, modules, arrow functions, and destructing assignments (ES6, 2015), leveraging its modular design. Subsequent yearly updates added features like simpler constructs for asynchronous programming (ES7, 2016), new math operators (ES8, 2017), and array/object syntax to enable destructuring, spread, and rest operations (ES9, 2018). This regular evolution was key to keep the language relevant in the software development landscape, allowing developers to leverage modern programming paradigms and best practices [10, 13].

As the language evolves to support better programming practices, developers need to adapt their codebases to take advantage of these new features. This process is known as *language rejuvenation* or *language migration*, and involves refactoring existing code to replace deprecated constructs with the new ones [13–16, 19]. Overall, software migration is a complex and error-prone task, often requiring significant effort from developers [1, 12, 18]. Many studies have investigated the use of Large Language Models (LLMs) to automate software migration tasks [1, 12]. For example, Almeida et al. [1] explored the use of ChatGPT to migrate two major versions of the SQLAlchemy library in one Python project. Islam et al. [12] assessed the performance of LLMs in migrating several Python libraries. Language migration is particularly challenging since it often involves structural changes to the codebase [15, 24].

However, we lack studies addressing the use of LLMs for language migration tasks. In this work, we aim to fill this gap by providing a dataset of real-world migrations from Promises to Async/Await syntax in JavaScript projects. We are particularly interested in this migration scenario due to two main reasons [10]. First, asynchronous programming is frequently used in JavaScript to circumvent its single-threaded design when developing responsive applications. Second, the Async/Await syntax greatly simplifies the management of asynchronous operations as it enables developers to write procedural-style code that is easier to read and maintain.

The PromiseAwait dataset contains 4,221 migration instances at its current version, extracted from 1,530 commits and 408 real-world projects. These instances were automatically detected using a heuristic based on regular expressions, manually validated by inspecting real migration cases on GitHub. This dataset represents an initial effort to build a benchmark for studying the performance of LLMs in performing language migration tasks.

***Data Availability.*** The PromiseAwait is publicly available at https://doi.org/10.5281/zenodo.17644182.

## 2 From Promises to Async/Await

Asynchronous programming is paramount in modern JavaScript development, allowing developers to implement responsive applications that can handle multiple tasks simultaneously [7, 10]. In the early days, JavaScript developers primarily relied on callbacks to manage asynchronous operations. However, as applications grew in complexity, this approach led to deeply nested and hard-to-maintain code structures, often referred to as "callback hell" [10].

This work is licensed under a Creative Commons Attribution 4.0 International License.
*FORGE '26, Rio de Janeiro, Brazil*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2477-0/2026/04
https://doi.org/10.1145/3793655.3793714

*The Advent of Promises.* The first significant improvement came with the introduction of Promises in ECMAScript 2015 (ES6) [8]. A promise represents a value computed asynchronously, and can assume one of three states: pending, fulfilled, or rejected [10]. Upon creation, a promise is in the pending state, waiting for the async operation to complete. Once the operation finishes, the promise is settled to either fulfilled (i.e., successfully completed) or rejected (i.e., failed). A `Promise` object is created with two parameters: `resolve` and `reject`. The first is called when the operation is successful, while the second is invoked in case of an error.

```
function fetchData() {
  return new Promise((resolve, reject) => {
      const data = { message: "Data fetched successfully!
           " };
      resolve(data);
    }, 2000);
  });
}

fetchData()
  .then((data) => console.log(data.message))
  .catch((error) => console.error("Error fetching data:",
        error));
```

**Figure 1: Example of using Promises in JavaScript**

Figure 1 illustrates a simple example of using Promises to handle an asynchronous operation. In this example, the `fetchData` function returns a promise that resolves after a 2-second delay. We then call `fetchData()`, and capture the resolved value using the `.then()` method, while errors are handled with the `.catch()` call. Although Promises significantly improved code readability compared to callbacks, they still present structural limitations, especially when chaining multiple asynchronous operations in sequence.

*The Emergence of Async/Await.* To improve developers' experience with asynchronous code, ECMAScript 2017 (ES8) introduced the *async/await* syntax [8]. This syntactic sugar allows developers to write and call asynchronous code as if it were synchronous, greatly enhancing code clarity and maintainability [10, 13]. All developers need to do is declare a function as `async`, which automatically makes it return a promise. Then, every call to an asynchronous operation can be prefixed with the `await` keyword, which pauses the function execution until the promise returned by the async operation is resolved or rejected.

```
async function fetchData() {
  const data = { message: "Data fetched successfully!" };
  return data;
}

try {
  const data = await fetchData();
  console.log(data.message);
} catch (error) {
  console.error("Error fetching data:", error);
}
```

**Figure 2: Example of using Async/Await in JavaScript**

Figure 2 presents the previous example refactored to use the async/await syntax. The `fetchData` is declared as an `async` function, thus it automatically wraps its return value in a promise. Also, the `fetchData()` call now is prefixed with the `await` keyword, which pauses execution until the promise returned by `fetchData` is resolved. In case the promise is rejected, the error is propagated to the surrounding `try...catch` block, which handles it appropriately. As we can see, the code is more idiomatic and easier to read, resembling synchronous code structure.

## 3 Data Collection

Our data collection process is organized in three major steps. This section explains each step in detail.

### 3.1 Repository Selection

On 10/22/2025, we fetched the top-500 most popular JavaScript projects from GitHub API, using the number of stars as popularity criterion [6]. For each project, the first author collected its name, URL, number of stars, forks, size, license, creation date, last commit date, and description. Next, the same author manually analyzed the title and description of each repository and filtered out those considered non-systems, i.e., educational repositories, tutorials, and toy examples. As a result, we ended up with 408 real-world JavaScript projects.

### 3.2 Detecting Async/Await Migration

*3.2.1 Inspecting Async/Await Migration Cases.* To better understand the changes performed when developers migrate from Promises to Async/Await, we manually analyzed pull requests created by developers on GitHub reporting this migration. Specifically, we searched on GitHub website for "Promise to async/await", and filtered by *pull requests* marked as *closed*, performed on *JavaScript* projects. We then manually investigated these PRs to identify key migration patterns and collected the source code of their repositories to use as test cases for heuristic development.

*3.2.2 Migration Heuristic Implementation.* Based on the manual inspection performed earlier, we implemented an automated heuristic to detect Promise to async/await migration events at scale. Given a source code diff, this heuristic relies on a set of regular expressions to detect the following patterns.

- *Removed code* snippets containing the Promise syntax keywords, specifically the keyword `Promise` and its chaining methods, such as `then()`.[1]
- *Added code* snippets containing `async` and *await* keywords, required for adopting Async/Await feature.[2]

Only source code diffs containing both criteria—Promise removal and async/await addition—were flagged as migration instances.

### 3.3 Collecting Migration Cases

Once we defined the heuristic for detecting Async/Await migration, we applied it to all commits performed across the projects initially selected. For each project, we cloned and traversed all commits performed on its main branch; we relied on PyDriller [21] for this

[1]`\bnew\s+Promise\s*\( |\.then\s*\(`
[2]`\basync\b|\bawait\b`

process. For each commit, we gathered all source code diffs and applied the heuristic to each one separately; the source code diffs were detected and matched using the UniDiff library.[3] We submitted each diff to our heuristic and selected the ones flagged as migration instances. In total, we analyzed commits from the 408 projects.

## 4 The PromiseAwait Dataset

*Dataset Characteristics.* The PromiseAwait dataset represents a first collection of real-world migrations from the Promise-based pattern to the async/await syntax in JavaScript projects. Currently, the dataset encompasses 4,221 migration instances, extracted from 1,530 commits and 134 projects. On average, each commit contains approximately 3.09 distinct instances, while projects average around 31.5 cases each.

Figure 3 presents the distribution of the migration instances. To keep this visualization clear, we excluded instances located at the last percentile, (i.e, removed the 42 instances with the highest line changes). We can see that the changes introduced by these instances are relatively small, since 87% added and removed no more than 100 lines. Interestingly, most migration instances removed more lines than they added. This suggests that the Async/Await syntax help reduce boilerplate code, leading to a more concise source code.

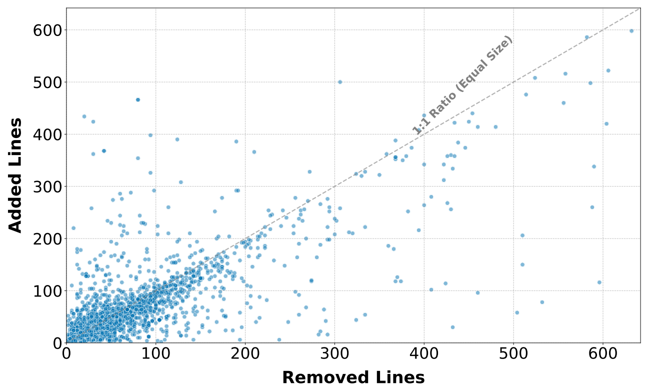


**Figure 3: Migration instances distributed according to their number of added and removed lines.**

*Dataset Structure.* At its current version, the PromiseAwait dataset is stored in a single CSV. Each line stores the metadata from the repository the migration belongs to, as well as the source code before and after the migration. Specifically, we maintain the following information for each migration instance.

- **Project:** Name of the project that was extracted.
- **Commit hash:** The unique SHA hash identifying the specific commit where the migration occurred.
- **Author:** Developer who authored the commit.
- **Message:** The full commit message, providing contextual information about the change.
- **File path:** The path to the specific file within the repository where the code change was made.
- **Commit url:** The direct URL link to the commit on GitHub, enabling easy external verification.
- **Removed chunk:** The block of code that was removed, containing the original Promise-based syntax.
- **Added chunk:** The block of code that was added, containing the new async/await syntax.
- **Commit date:** Date and time of the authored commit.

## 5 Migration Examples

To give readers a concrete view of the migrations collected in the PromiseAwait, Figure 4 depicts two real-world examples of Promise to Async/Await migrations.

*Simple Example.* Figure 4a illustrates a straightforward migration case implemented on the **CesiumJS** project, a widely-used open-source JavaScript library for creating 3D globes and maps. This migration refactors a function responsible for loading metadata schemas asynchronously. The original code relies on a Promise chain, invoking `schemaLoader.load()` and handling the subsequent action within a `.then()` function. The code inside this call, executed after the Promise triggered by `schemaLoader.load()` is resolved, initializes a metadata object containing the loaded schema. Finally, this object is assigned to the `tileset._metadataExtension` property, declared in the outer scope of the chain.

The migrated code removes the entire Promise chain, allowing the scope of asynchronous operations to be shared with the rest of the function. The `metadataExtension` initialization can now rely on the result of the `await schemaLoader.load()` call. As this object now belongs to the same scope as the rest of the function, it can be directly returned at the end of the function, without needing to be assigned to an outer-scoped variable first. All in all, this migration significantly simplifies the code structure, making it more linear and easier to follow.

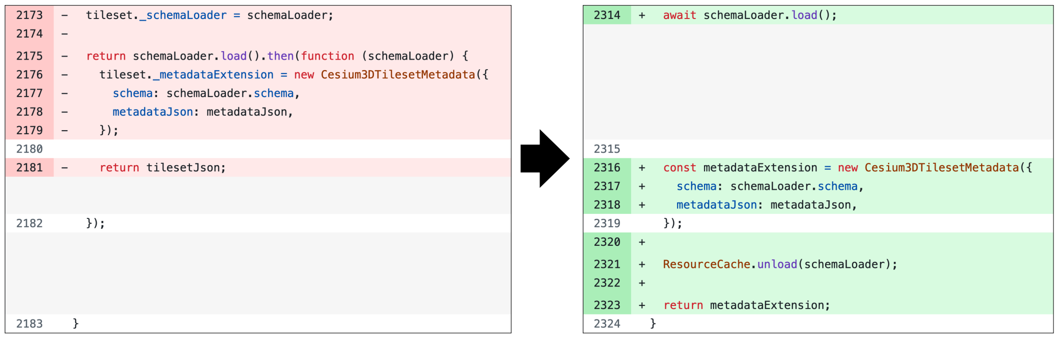
*Complex Example.* Figure 4b presents a migration case from the **Codesandbox-client** project, a popular Integrated Development Environment (IDE) for web development. This transition involves refactoring a fetch operation that retrieves and processes external data. The original code employs a Promise chain, starting with a `fetch()` call, followed by a sequence of `.then()` calls to sequentially process the response and parse it as a JSON object. This chain ensures that each step is executed only after the previous one has resolved successfully. Finally, a `.catch()` block is used to handle any errors that may occur during the fetch or processing steps.

The migrated code replaces the whole Promise chain with a `try...catch` block. Inside the `try` block, the fetch operation and subsequent JSON parsing are handled sequentially using `await` calls. If any error occurs during these operations, it is caught by the surrounding `catch` block. In practice, the refactored code can be interpreted sequentially, resembling synchronous code structure.

## 6 Related Work

*Library Migration.* This transition refers to replacing one library with another in a software project [11, 12]. Prior studies have explored this phenomenon in depth. Barbosa and Hora [3] examined migrations from *unittest* to *pytest* in open-source Python projects, highlighting motivations such as new features, simpler syntax, and

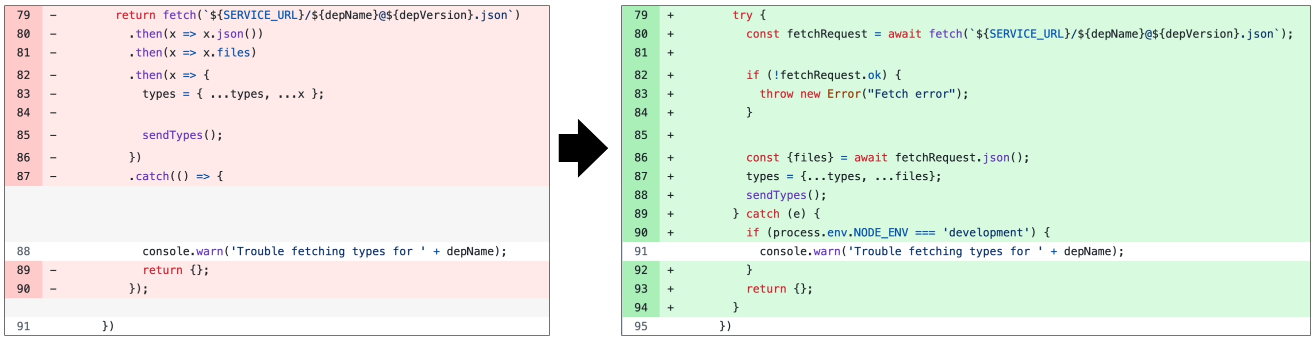
[3]https://pypi.org/project/unidiff/



```
-   tileset._schemaLoader = schemaLoader;
-
-   return schemaLoader.load().then(function (schemaLoader) {
-     tileset._metadataExtension = new Cesium3DTilesetMetadata({
-       schema: schemaLoader.schema,
-       metadataJson: metadataJson,
-     });

-     return tilesetJson;

    });
  }
```

```
+   await schemaLoader.load();

+   const metadataExtension = new Cesium3DTilesetMetadata({
+     schema: schemaLoader.schema,
+     metadataJson: metadataJson,
    });
+
+   ResourceCache.unload(schemaLoader);
+
+   return metadataExtension;
  }
```

(a) Simple example, performed on CesiumJS.



```
-         return fetch(`${SERVICE_URL}/${depName}@${depVersion}.json`)
-           .then(x => x.json())
-           .then(x => x.files)
-           .then(x => {
-             types = { ...types, ...x };
-
-             sendTypes();
-           })
-           .catch(() => {

              console.warn('Trouble fetching types for ' + depName);
-             return {};
-           });

        })
```

```
+         try {
+           const fetchRequest = await fetch(`${SERVICE_URL}/${depName}@${depVersion}.json`);
+
+           if (!fetchRequest.ok) {
+             throw new Error("Fetch error");
+           }
+
+           const {files} = await fetchRequest.json();
+           types = {...types, ...files};
+           sendTypes();
+         } catch (e) {
+           if (process.env.NODE_ENV === 'development') {
              console.warn('Trouble fetching types for ' + depName);
+           }
+           return {};
+         }
        })
```

(b) A complex example, performed on Codesandbox-client.

**Figure 4: Source code examples illustrating more simple and complex Promise-to-Async/Await migrations.**

flexibility; Alves and Hora [2] extended this work by curating a dataset of corresponding source code migrations. Islam et al. [12] characterized structural changes in 311 Python projects, identifying patterns such as the use of decorators and exception handling. More broadly, Gu et al. [11] compared migration practices across Java, JavaScript, and Python ecosystems, showing common motivations like deprecation or feature improvement.

*Language Migration.* By contrast, language migration focuses on adapting the codebase to support new language features [4, 10, 14, 16, 19]. Developers are compelled to perform these transitions to increase code readability, conciseness, and maintainability [13, 20]. Previous studies have investigated this phenomenon in various contexts. Malloy and Power [14] quantified to what extent open-source Python projects migrated from Python 2 to Python 3, concluding that developers were reluctant to perform this migration due to compatibility issues. Mendonça et al. [16] surveyed Java developers about their impressions on adopting lambda expressions in the language, finding this feature improves code readability when it replaces anonymous classes and structural loops. Other studies explored migrations between different languages. Gokhale et al. [10] analyzed the transition from synchronous to asynchronous programming in JavaScript, highlighting the challenges developers face when adapting to this paradigm shift. In this work, we focused on providing a dataset with real-world migrations between two asynchronous programming solutions in JavaScript: Promises and Async/Await syntax.

## 7 Conclusion

In this work, we presented PromiseAwait, a dataset of real-world migrations from Promises to Async/Await syntax in JavaScript projects. At its current state, the dataset is composed of 4,221 migration instances, obtained from 134 real-world projects. Each instance contains the isolated code change that performed the migration, allowing researchers to easily evaluate the performance of LLMs in this specific migration task.

*Next Steps.* We intend to extend this work in two main directions. First, we plan to expand the dataset with different migration scenarios, such as the transitions from callbacks to Promises, and from traditional function expressions to arrow functions. Second, we aim to use this dataset as a benchmark to evaluate the performance of LLMs in performing language migration tasks.

## Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. It was also supported by grants from CNPq (403304/2025-3) and by FAPEMIG (APQ-02419-23).

## References

[1] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 2024. Automatic Library Migration Using Large Language Models: First Results. In *18th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–7.

[2] Altino Alves and Andre Hora. 2025. TestMigrationsInPy: A Dataset of Test Migrations from Unittest to Pytest. In *Mining Software Repositories (MSR): Data and Tools Showcase Track*. 1–5.

[3] Livia Barbosa and Andre Hora. 2022. How and Why Developers Migrate Python Tests. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 538–548.

[4] Luciano Baresi, Massimiliano Di Penta, Giovanni Quattrocchi, and Damian Andrew Tamburri. 2024. How Have iOS Development Technologies Changed over Time? A Study in Open-Source. In *Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems*. 33–42.

[5] Justus Bogner and Manuel Merkel. 2022. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and Typescript Applications on GitHub. In *19th International Conference on Mining Software Repositories (MSR)*. 658–669.

[6] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* (2018), 112–129.

[7] Douglas Crockford. 2017. *JavaScript: The Good Parts: The Good Parts*. O'Reilly Media.

[8] ECMA International. 2025. ECMAScript Language Specification.

[9] Fabio Ferreira, Hudson Borges, and Marco Tulio Valente. 2024. Refactoring React-based Web Apps. *Journal of Systems and Software* 1 (2024), 1–36.

[10] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic Migration from Synchronous to Asynchronous JavaScript APIs. *Proceedings of the ACM on Programming Languages* OOPSLA (2021), 1–27.

[11] Haiqiao Gu, Hao He, and Minghui Zhou. 2023. Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 627–638.

[12] Mohayeminul Islam, Ajay Kumar Jha, Ildar Akhmetov, and Sarah Nadi. 2024. Characterizing Python Library Migrations. In *ACM International Conference on the Foundations of Software Engineering (FSE)*, Vol. 1. 1–23.

[13] Walter Lucas, Rafael Nunes, Rodrigo Bonifácio, Fausto Carvalho, Ricardo Lima, Michael Silva, Adriano Torres, Paola Accioly, Eduardo Monteiro, and João Saraiva. 2025. Understanding the Adoption of Modern Javascript Features: An Empirical Study on Open-Source Systems. *Empirical Software Engineering* (2025), 1–42.

[14] Brian A. Malloy and James F. Power. 2017. Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 314–323.

[15] Walter Mendonça. 2024. Towards a Theory for Source Code Rejuvenation. In *32nd ACM International Conference on the Foundations of Software Engineering (FSE)*. 701–703.

[16] Walter Lucas Monteiro Mendonça, José Fortes, Francisco Vitor Lopes, Diego Marcílio, Rodrigo Bonifácio De Almeida, Edna Dias Canedo, Fernanda Lima, and João Saraiva. 2020. Understanding the Impact of Introducing Lambda Expressions in Java Programs. *Journal of Software Engineering Research and Development* 8 (2020).

[17] João Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying Experts in Software Libraries and Frameworks Among GitHub Users. In *16th International Conference on Mining Software Repositories (MSR)*. 276–287.

[18] João Eduardo Montandon, Luciana Lourdes Silva, Cristiano Politowski, Daniel Prates, Arthur de Brito Bonifácio, and Ghizlane El Boussaidi. 2025. Unboxing Default Argument Breaking Changes in 1 + 2 Data Science Libraries. *Journal of Systems and Software* (2025), 1–38.

[19] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and Use of Java Generics. *Empirical Software Engineering* (2013), 1047–1089.

[20] Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Alexandre Bergel, and Nicolas Anquetil. 2015. Does JavaScript Software Embrace Classes?. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 73–82.

[21] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. *PyDriller: Python Framework for Mining Software Repositories*.

[22] Stack Overflow. 2024. Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/.

[23] GitHub Staff. 2025. Octoverse: A New Developer Joins GitHub Every Second as AI Leads TypeScript to #1.

[24] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. 2025. Migrating Code At Scale With LLMs At Google. In *ACM International Conference on the Foundations of Software Engineering (FSE)*. 1–12.