Noname manuscript No.
(will be inserted by the editor)

Understanding the Adoption of Modern JavaScript Features: An Empirical Study on Open-Source Systems

Walter Lucas · Rafael Nunes · Rodrigo Bonifácio · Fausto Carvalho · Ricardo Lima · Michael Silva · Adriano Torres · Paola Accioly · Eduardo Monteiro · João Saraiva

the date of receipt and acceptance should be inserted later

Lucas, Walter
University of Brasília, Brasília, Brazil
E-mail: waltimlmm@gmail.com

Nunes, Rafael
University of Brasília, Brasília, Brazil
E-mail: nunes.rafael@aluno.unb.br

Bonifácio, Rodrigo
University of Brasília, Brasília, Brazil
E-mail: rbonifacio@unb.br

Carvalho, Fausto
University of Brasília, Brasília, Brazil
E-mail: faustocarva@gmail.com

Lima, Ricardo
University of Brasília, Brasília, Brazil
E-mail: ricdelima@gmail.com

Silva, Michael
University of Brasília, Brasília, Brazil
E-mail: michaeldf.ti@gmail.com

Adriano, Torres
University of Brasília, Brasília, Brazil
E-mail: adrianortorres@gmail.com

Accioly, Paola
Federal University of Pernambuco, Brazil
E-mail: prga@cin.ufpe.br

Eduardo, Monteiro
University of Brasília, Brasília, Brazil
E-mail: edumonteiro@unb.br

Saraiva, João
University of Minho, Braga, Portugal
E-mail: saraiva@di.uminho.pt

Abstract JavaScript is a widely used programming language initially designed to make the Web more dynamic in the 1990s. In the last decade, though, its scope has extended far beyond the Web, finding utility in backend development, desktop applications, and even IoT devices. To circumvent the needs of modern programming, JavaScript has undergone a remarkable evolution since its inception, with the groundbreaking release of its sixth version in 2015 (ECMAScript 6 standard). While adopting modern JavaScript features promises several benefits (such as improved code comprehension and maintenance), little is known about which modern features of the language have been used in practice (or even ignored by the community). To fill this gap, in this paper, we report the results of an empirical study that aims to understand the adoption trends of modern JavaScript features, and whether or not developers conduct rejuvenation efforts to replace legacy JavaScript constructs and idioms with modern ones in legacy systems. To this end, we mined the source code history of 158 JavaScript open-source projects, identified contributions to rejuvenate legacy code, and used time series to characterize the adoption trends of modern JavaScript features. The results of our study reveal extensive use of JavaScript modern features which are present in more than 80% of the analyzed projects. Our findings also reveal that (a) the widespread adoption of modern features happened between one and two years after the release of ES6 and, (b) a consistent trend toward increasing the adoption of modern JavaScript language features in open-source projects and (c) large efforts to rejuvenate the source code of their programs.

Keywords Software Evolution, Software Engineering, Software Maintenance, Source Code Rejuvenation, Mining Software Repositories

1 Introduction

Originally designed to enhance interactivity on Web pages, JavaScript has evolved into a versatile language used across web, mobile, and server-side development. Different JavaScript environments are available today to support a wide range of applications, including backend and desktop development, as well as emerging technologies such as smart contracts and IoT. Additionally, JavaScript serves as a compilation target for languages such as Reason and TypeScript, enabling their code to run in JavaScript engines. It also underpins rendering frameworks like Vue.js, ReactJS, and many others (Silva et al., 2015).

By 1996, the JavaScript standardization process was initiated under Ecma International, leading to the first official ECMAScript specification in 1997 (Ecma, 1997). Two additional revised editions were released by the end of 1999, reflecting Netscape's enhancements to the language. However, the early 2000s saw a period of stagnation. In 2008, the committee restored its operations, resulting in a modestly enhanced edition published in 2009. This success paved the way for the major enhancements seen in ECMAScript 2015 (ES6),

which laid the foundation for JavaScript's recent evolution (Wirfs-Brock and Eich, 2020; Flanagan, 2020).

Post-2015, the committee adopted a process for faster, incremental releases, completing revisions on a yearly schedule. This structured approach has allowed JavaScript to continuously adapt and grow, maintaining its dominance in web development, where the evolution of the language has introduced numerous features—such as ES6 modules, asynchronous programming with Promises, and syntax improvements (Wirfs-Brock and Eich, 2020).

However, adapting legacy code to newer JavaScript features is challenging and involves trade-offs between various factors, including the benefits of new features, project requirements, and compatibility with target environments (e.g., JavaScript engines like V8) (Nicolini et al., 2024). Even more importantly, rejuvenating old code to leverage new language features and idioms can require a substantial time investment—although previous research suggests that the expected benefits of source code rejuvenation pay off (Lucas et al., 2023).

Previous research has investigated the practices developers follow to rejuvenate their code and understand the motivations and challenges behind these efforts and trends for adopting programming language features. Albeit focusing on statically typed languages such as Java (Dyer et al., 2014a; Mazinanian et al., 2017; Dantas et al., 2018; Lucas et al., 2019; Zheng et al., 2021), C++ (Kumar et al., 2012; Uesbeck et al., 2016; Chen et al., 2020; Lucas et al., 2023), Kotlin (Mateus and Martinez, 2020), and TypeScript (Scarsbrook et al., 2023). For instance,Mazinanian et al. investigate the adoption of lambda expressions in Java, particularly focusing on how developers transition to functional programming, while (Lucas et al., 2023) show that C++ software developers contributing to KDE projects often conduct large source code rejuvenation efforts.

Despite the widespread use of the JavaScript programming language, relatively few studies have explored the efforts involved in adopting modern JavaScript features, as well as the motivations and challenges driving these efforts (Silva et al., 2015; Gokhale et al., 2021; Alves et al., 2022; Nicolini et al., 2024). This paper aims to advance the existing literature by addressing these gaps through the findings of an empirical study that analyzes the source code history of open-source JavaScript projects, shedding light on the adoption and rejuvenation efforts in this context.

Accordingly, we investigate (a) the adoption trends of modern JavaScript (JavaScript) features, (b) the extent of developers' efforts to rejuvenate legacy JavaScript source code, and (c) the motivations driving developers to rejuvenate legacy JavaScript code, as well as the challenges that hinder such efforts. Our study evaluated the adoption of modern JavaScript features in 158 open-source GitHub repositories created at least three years before the release of ES6, a landmark in modern JavaScript. Here, we provide an in-depth study on the adoption of modern JavaScript features in legacy open-source code, highlighting:

- **Early Adoption of Features**: Some modern JavaScript features are utilized even before their official release, reflecting developers' anticipation and readiness to embrace advancements in the language.
- **Variation in Feature Popularity**: While some features seem quite popular among the projects in our dataset, such as Const Declarations, Async Declarations, and Arrow Function Declarations that appear in more than 80% of the projects, others seem not that popular (like Computed Property Assignment from ES6 that appear in less than 35% of the projects).
- **Motivations for Adoption:** After analyzing code review discussions on source code rejuvenation, we identified several motivations for adopting modern features, including improved code readability, conciseness, and maintainability. These motivations align with trends observed in previous studies on statically typed languages (e.g., Java (Mazinanian et al., 2017) and Kotlin (Mateus and Martinez, 2020)) and a prior study on JavaScript that examines the adoption of classes (Silva et al., 2015). In contrast, our study focuses on a broader set of JavaScript features.
- **Challenges in Adoption**: Compatibility issues, resistance to change, and performance concerns present significant barriers, underscoring the importance of strategies to mitigate these challenges.
- **Code Rejuvenation Practices**: Developers actively refactor and update existing codebases to incorporate modern features after their releases, emphasizing the need for tools that assist in identifying and automating these transformations.

This paper is organized as follows: Sections 2 present some background information about the evolution of JavaScript language. In Section 3 we detail our study settings. Section 4 present the first study, which used descriptive statistics and time series to examine the adoption trends of modern JavaScript features, Section 5 present a second study, which used thematic analysis of code review comments to highlight the motivations and challenges of updating JavaScript code. Section 6 present more details about our paper findings and implications. Section 7 compare our study with previous work about source software rejuvenation, adoption trends, motivations, and challenges from diverse programming languages in the literature. Finally, Section 8 present our conclusions and future works.

2 Background

In this section, we introduce the concept of source code rejuvenation (Section 2.1) and provide a brief overview of the evolution of the JavaScript language (Sections 2.2 and 2.3), highlighting the introduction of modern features. Section 2.4 then discusses the challenges of JavaScript cross-version compatibility and its potential impact on the rapid adoption of new language features.

2.1 Source Code Rejuvenation

Source code rejuvenation refers to the process of updating and transforming legacy code to incorporate modern programming language features and idioms (Pirkelbauer et al., 2010). Unlike general maintenance tasks, this approach focuses specifically on replacing outdated constructs and idioms with contemporary equivalents to align with advancements in language design. By addressing inconsistencies and legacy patterns, source code rejuvenation improves code comprehension and maintainability, providing developers with a more cohesive and up-to-date codebase (Lucas et al., 2019, 2023).

For instance, modern JavaScript introduces Const Declarations and Let Declarations as block-scoped variable declarations that replace the function-scoped 'var', and Arrow Function Declarations as a concise syntax for defining anonymous functions (Rauschmayer, 2015). Using Const Declarations ensures variables are immutable by default and enforces block scope, reducing the risk of unintended variable overwrites and improving code clarity. Figure 1 shows an example of a rejuvenation effort to adopt Const Declarations.

On left side, the legacy code uses `var`, which allows variable reassignment and does not enforce block scope. As a result, the variable `isAdult`, declared inside the `if` block, remains accessible outside of it. In contrast, the right side presents the rejuvenated code, where `var` is replaced with `const`, making variables immutable and ensuring they are confined to their respective blocks (Rauschmayer, 2015). This prevents unintended modifications and scoping issues. Consequently, attempting to access `isAdult` outside its block results in a `ReferenceError`.

```
var userName = "Alice";
var userAge = 30;
if (userAge > 18) {
    var isAdult = true;
}
console.log(isAdult);
```

```
const userName = "Alice";
const userAge = 30;
if (userAge > 18) {
    const isAdult = true;
    console.log(isAdult);
}
console.log(isAdult);
```

Fig. 1 Replacing `var` with `const`. The code on the left uses the `var` construct, while the code on the right uses the new `const` feature.

Arrow Function Declarations provide a shorter syntax and lexically bind the `this` context, making them particularly useful for callbacks and functional programming paradigms (Rauschmayer, 2015). By systematically applying such transformations, developers can align codebases with modern JavaScript standards. On the left side, the legacy code defines a function using a traditional function expression, which requires an explicit `return` statement. In contrast, the right side showcases the rejuvenated version, where an arrow function replaces the function expression, reducing verbosity.

```
function add(a, b) {
    return a + b;
}
console.log(add(1, 2));
```

```
const add = (a, b) => a + b;
console.log(add(1, 2));
```

Fig. 2 Replacing function expressions definitions with Arrow Function Declarations. The code on the left uses the a conventional syntax for implementing functions; while the code on the right use the Arrow Function Declarations feature introduced in ES6.

In previous work, Pirkelbauer and colleagues advocate that source code rejuvenation is not refactoring (Pirkelbauer et al., 2010). The main reason is that, unlike refactoring, source code rejuvenation focuses specifically on replacing outdated constructs and idioms with contemporary equivalents to align with advancements in language design. In this paper, we prefer not to use the term refactoring for rejuvenation efforts, in particular, because the kinds of transformations we discuss here do not appear in refactoring references for JavaScript (Silva et al., 2021; Brito et al., 2022) and other programming languages as Java (Tsantalis et al., 2018).

2.2 Historical JavaScript

The JavaScript programming language, more formally known as ECMAScript (ES for short), was initially designed to make the World Wide Web more dynamic. Currently, it is widely used in many other fields of software development besides the Web, although it was not originally conceived with that intent. Brendan Eich led the initial design of the JavaScript language, which at the time was referred to as Mocha. After first appearing on Netscape Navigator, it went through significant changes, evolving from LiveScript to its third version and eventually being named JavaScript. This nomenclature change reflected the integration of certain syntactic elements inspired by the Java programming language (Wirfs-Brock and Eich, 2020).

To relieve the problems of having different JavaScript flavors, the Netscape company asked the European Computer Manufacturers Association (ECMA) to standardize the JavaScript syntax and semantics as a general-purpose, cross-platform, vendor-neutral scripting language. This effort led to the ECMA-262 standard, and other browsers that aim to benefit from JavaScript could use the specification to implement a JavaScript runtime (Keith, 2005). Nowadays, JavaScript is ubiquitous and has significantly changed in the past three decades of development. At the time of this writing, there are 14 official releases of the language, broadly used to develop frontend and backend components for web, mobile, and desktop applications, ranging from REST-based applications to more sophisticated ones. On the backend, JavaScript is mainly used with a runtime (such as NodeJS) that exposes APIs commonly found in other programming languages. The initial standardized version of JavaScript (ES1 standard) supported fundamental imperative programming features such as variables and function declarations, as well as conditional and repetitive statements (Wirfs-

Brock and Eich, 2020). In the subsequent versions, ECMAScript versions 2 and 3, the language was extended to include support for error handling, enhanced the support for regular expressions, and introduced new array manipulation capabilities (Wirfs-Brock and Eich, 2020). Curiously, JavaScript version 4 was never published, as its development was stalled due to internal discussions about the future directions of the language (Wirfs-Brock and Eich, 2020).

The ES5 standard, released in 2009, introduced significant improvements. It supports additional capabilities for string manipulation and the implementation of the JSON (Javascript Object Notation) data (de) serialization; object-oriented programming features; and a functional style for array manipulation, including methods such as `reduce()`, `filter()`, and `map()`(Wirfs-Brock and Eich, 2020; Resig et al., 2016; Flanagan, 2020).

Next we introduce more recent JavaScript releases, collectively referred to as "Modern JavaScript."

2.3 Modern JavaScript

The release of the ES6 standard in 2015 corresponds to a significant milestone in the evolution of JavaScript. ES6 introduces the support for the repetition statement `for ...of` on *iterable objects*; template literals; rest parameters (a.k.a., variadic functions in other languages); and lambda expressions (i.e., arrow functions). Additionally, ES6 also enhances the decomposition of JavaScript programs into modules, fostering a modular approach to code organization and reuse (Resig et al., 2016).

These additional features collectively enhanced the language's capabilities for programming in different paradigms (e.g., object-oriented programming and functional programming) (Wirfs-Brock and Eich, 2020; Flanagan, 2020), increased the expressivity of the language, and contributed to improving code readability and the contextual separation of code.

ES6 also introduced the `let` and `const` keywords for block-scoped variables, destructuring assignments for concise variable assignments, improved iterators and generators to facilitate the implementation of control flow, and `Promises`, a mechanism for implementing asynchronous operations (Resig et al., 2016; Flanagan, 2020). Indeed, the modifications in this version were so substantial that the term *"Modern JavaScript"* is now commonly associated with code written using ES6 syntax and beyond (Morgan and Stewart, 2018).

From 2016 to 2020, ECMAScript versions adopted a yearly release cycle, shifting from numerical versioning (e.g., ES5, ES6) to a naming convention based on the year of publication (e.g., ES2016, ES2017). During this period, ES2016 to ES2020 introduced new language features while refining existing ones, following the standardization efforts of Technical Committee 39 (TC39)—the committee responsible for standardizing the JavaScript programming language. These versions consistently reviewed existing features throughout this period and introduced novel constructs (Wirfs-Brock and Eich, 2020).For instance, in 2016, ES2016 introduced (a) the array method

includes(), which verifies whether an array contains a given value, and (b) the exponentiation operator (**) for mathematical calculations. In 2017, ES2017 introduced the `async/await` keywords, simplifying asynchronous programming by allowing developers to write asynchronous code in a more readable, synchronous-like style (Flanagan, 2020).

The release of ES2018 in 2018 introduced the spread operator for copying the properties of an existing object into a new object, along with the addition of the `for-await-of` loop, designed to work with asynchronous iterators (Flanagan, 2020). In 2019, the language standard ES2019 introduced new methods for array manipulation (`Array.prototype`), such as `flat()` and `flatMap()`. Additionally, ES2019 standardized the `JSON.stringify()` behavior for `BigInt` values and refined various syntax and semantic rules (Flanagan, 2020). These enhancements further enriched the language with additional functional programming features. In 2020, ES2020 introduced the `BigInt` data type, the `globalThis` object, the `nullish coalescing operator` (`??`) for providing default values, and `optional chaining` (`?.`) for safe property access. These new language features aim to simplify common programming patterns and provide a more concise syntax (Flanagan, 2020).

ES2021 (ES2021) introduced several enhancements, including numeric separators, which improve program readability by allowing underscore characters within numeric literals. ES2021 also introduced the `replaceAll()` method, providing a direct way to replace all occurrences of a substring in a string. Additionally, ES2021 added the `WeakRef` and `FinalizationRegistry` APIs, enabling the garbage collector to manage weak references and perform cleanup operations more effectively. Furthermore, ES2021 introduced the `Promise.any()` combinator, which resolves as soon as the first promise in an array fulfills, streamlining the handling of multiple asynchronous operations(Kelhini, 2021).

ES2022 (ES2022) introduced several syntax and library enhancements to improve code structure and usability. Among the syntax changes, it introduced class fields and private methods, enabling developers to define properties and methods with restricted access directly within class definitions, without relying on constructor-based initialization. Additionally, ES2022 introduced static blocks in classes, allowing static initialization logic to be executed once per class definition. In terms of library updates, ES2022 added the `Array.prototype.at()` method, which provides a more convenient way to access elements from an array using positive or negative indices. Furthermore, it introduced top-level `await` in modules, eliminating the need for wrapping asynchronous logic within an async function at the module level, thereby simplifying asynchronous code execution (Phang, 2022).

ES2023 (ES2023) and ES2024 (ES2024) primarily introduced updates to built-in libraries, enhancing functionality and improving language consistency. ES2023 expanded the `at()` method to support Strings and TypedArrays, providing a more intuitive way to access elements using relative indexing. It also introduced the *cause* property for `Error` objects, allowing developers to track error causation chains more effectively. Additionally, ES2023 refined module

handling with top-level `await`, enabling asynchronous operations at the module level without requiring an enclosing async function. ES2024 continued this trend of library enhancements by improving standard objects and refining existing methods, reinforcing JavaScript's commitment to usability and robustness (Rauschmayer, 2024). Our study focused on ECMAScript versions up to ES2023, as data collection was completed by December 31, 2023.

2.4 JavaScript Cross-Version Compatibility

The issue of JavaScript cross-version compatibility is central to understanding the factors that might hinder developers from rejuvenating JavaScript code. Backward compatibility is a key JavaScript design constraint, ensuring that older code remains functional on newer JavaScript engines (Andreasen et al., 2017).

However, language modifications introduce incompatibilities in some scenarios, particularly when new features repurpose previously undefined keywords. Python addresses this challenge with the concept of *soft keywords*, introduced in Python 3.10 to enable the Structural Pattern Matching feature. While JavaScript does not formally implement soft keywords, it achieves a similar effect through context-sensitive parsing, where certain keywords (e.g., await) are only treated as reserved within specific contexts.

JavaScript runs in a wide range of environments, including application servers, IoT devices, and blockchain applications. Nevertheless, these uses of JavaScript often depend on JavaScript engines originally developed for popular browsers. For example, Node.js uses the V8 engine from Google Chrome. Therefore, understanding how quickly browsers adopt new versions of the JavaScript language also sheds light on JavaScript adoption trends in other scenarios. Major JavaScript engines, such as V8, SpiderMonkey (Firefox), and JavaScriptCore (Safari and Bun), tend to implement features at a similar pace across platforms [1], though modern JavaScript features achieve full compatibility in browsers at varying rates.

For instance, the ES6 version was released in 2015 and achieved broad browsers support by 2017, while features from ES2018 and ES2019 required additional years, reaching full compatibility by 2020. Surprisingly, although ES2021 was officially released in 2021, most browsers offer full support to its features already to 2020.

This apparent discrepancy—where the "Release Year" and "Browsers Full Support" dates do not coincide—reflects the reality of progressive feature integration by browser vendors. In other words, while the formal specification was published later, many engines had already incorporated its features before the official release. Such gradual and sometimes anticipatory adoption underscores the ongoing standardization challenge across browsers, which in turn influences the broader adoption trends of JavaScript in various environments [2].

[1] https://compat-table.github.io/compat-table/es2016plus/

[2] https://www.w3schools.com/js

It is important to highlight that the compatibility challenge varies between domains: browser environments require meticulous handling to accommodate a broad spectrum of user configurations, whereas server-side developers have more control over runtime environments, enabling them to adopt newer features more aggressively. Additionally, the evolution of JavaScript features is guided by a formal process outlined in the TC39 Process Document [3], which explains the different stages that proposed features must go through. This process is crucial because JavaScript engines only implement features that have reached either the third or the fourth stage, ensuring stability and interoperability across environments.

Transitioning to newer JavaScript versions presents challenges related to code adjustments (Gokhale et al., 2021; Alves et al., 2022). For instance, the migration to ES6 (ECMAScript 2015) turned JavaScript into a multi-paradigm language, introducing classes, arrow functions, and promises. To ensure compatibility with older engines, developers often use transpilers, e.g., Babel (Nicolini et al., 2024), which transform modern code into versions that older engines can execute. However, despite these tools, execution failures may still occur in unsupported environments.

3 Overall Research Design

Our study aims to build an understanding of (a) modern JavaScript features adoption (from ES6 onwards) in legacy open-source projects and (b) the potential benefits and challenges associated with JavaScript source code rejuvenation. Our analysis considers 32 modern JavaScript features (including Object Destructuring, Arrow Function Declarations, and Async Declarations) introduced in ECMAScript versions from 2015 to 2022 that modify the language's syntax. Therefore, we focus on new syntactic elements of the JavaScript language, excluding from our analysis changes to the standard libraries, such as enhancements introduced in the regular expression library of modern JavaScript. Table 1 provides a comprehensive overview of modern JavaScript features included in our study. The "Feature" column lists each specific language capability. The "JavaScript Version" column indicates the ECMAScript version in which each feature was formally introduced. The "Release Year" column shows the official release year of the respective ECMAScript version, offering a temporal context for when these features became part of the standardized language. Lastly, the "Simple Example" column presents a concise code snippet that demonstrates the usage of the feature, making it easier for readers to understand its practical application. To achieve the goals of our study, we address the following research questions:

(RQ1) To what extent do JavaScript open-source repositories rely on modern features?

[3] https://tc39.es/process-document/

(RQ2) When did JavaScript developers start using each one of the modern JavaScript features?
(RQ3) What are the adoption trends of each one of the modern JavaScript features?
(RQ4) What are the main motivations that lead developers to rejuvenate legacy JavaScript code, and what are the main challenges that prevent them from doing so?

Table 1 Modern JavaScript Features, ECMAScript Version, Official Release Year, and Simple Usage Examples.

Feature	JavaScript Version	Official Release Year	Simple Example
Const Declarations	ES6	2015	`const x = 10;`
Let Declarations	ES6	2015	`let y = 5;`
Object Destructuring	ES6	2015	`const {a, b} = obj;`
Array Destructuring	ES6	2015	`const [a, b] = arr;`
Yield Operators	ES6	2015	`function* gen(){yield 1;}`
Enhanced Property Assignment	ES6	2015	`const obj = {x, y};`
Arrow Function Declarations	ES6	2015	`const f = ()=> {};`
Default Parameters	ES6	2015	`function f(x = 1){}`
Class Declarations	ES6	2015	`class MyClass {}`
Import Statements	ES6	2015	`import {x} from 'y';`
Export Declarations	ES6	2015	`export const x = 1;`
Rest Statements	ES6	2015	`function f(...args){}`
Template String Expressions	ES6	2015	`const s = 'Hello ${n}';`
Function Property Declarations	ES6	2015	`const obj = {f(){}};`
Spread Arguments	ES6	2015	`const arr = [...iterable];`
Computed Property Assignment	ES6	2015	`const obj = {[key]: value};`
For of Statements	ES6	2015	`for (const item of array){}`
Exponentiation Assignments	ES2016	2016	`x **= 2;`
Async Declarations	ES2017	2017	`async function foo(){}`
Await Operators	ES2017	2017	`await asyncFunction();`
For Await Of Statements	ES2018	2018	`for await (const x of xs){}`
Spread in Objects	ES2018	2018	`const obj = {...otherObj};`
Rest in Objects	ES2018	2018	`const {a, ...rest} = obj;`
Optional Catch Binding	ES2019	2019	`try {} catch {}`
Optional Chain	ES2020	2020	`const value = obj?.prop;`
Null Coalesce Operators	ES2020	2020	`const value = a ?? b;`
BigInt	ES2020	2020	`const big = 123n;`
Numeric Separator	ES2021	2021	`const num = 1_000_000;`
Assignment Operators	ES2021	2021	`x ||= y;`
Static Block in Classes	ES2022	2022	`class C { static { }};`
Private Fields	ES2022	2022	`class MyClass { #x; }`
Private Methods	ES2022	2022	`class MyClass { #m(){}};`

Answering the first research question helps us to understand to what extent JavaScript developers *embrace* modern JavaScript features and whether the new releases of the language are meant for the development of JavaScript applications. For example, arrow functions are anticipated to enhance developer expressiveness. Given that JavaScript functions are treated as first-class citizens, it is expected that developers would extensively utilize arrow functions. Conversely, JavaScript classes may not be as appealing since JavaScript already offers encapsulation through prototypes (Haverbeke, 2014).

Answering the second question helps us understand when developers start to adopt modern JavaScript features. The second question explores how confident developers are to switch from already established constructs and idioms to new ones that might eventually increase productivity, code conciseness,

and even lead to security improvements. Exploring the third question helps us evaluate whether there is a trend of increasing adoption of modern JavaScript features in open-source projects.

JavaScript is known for its rapid iteration cycle when developing applications, but the question of how that reflects on maintenance efforts to rejuvenate legacy code is still open. The fourth question focuses on the motivations for JavaScript code rejuvenation and investigates the challenges developers face that prevent them from rejuvenating legacy systems.

3.1 Projects Selection Procedures

Our approach to project selection was inspired by the work of Zhao et al. (Zhao et al., 2017). In the first step, we searched for GitHub repositories using JavaScript as the main programming language, specifically targeting those created in or before 2012 and active in 2023. We chose projects that were at least three years old by 2015 to focus on potentially mature and legacy systems, as systems that have been operational for at least a couple of years are likely to face challenges related to technological obsolescence and misalignment with current practices (Rosenkranz et al., 2024). This decision allows us to investigate projects that evolved using the "pre-modern" JavaScript language, relying on outdated constructs and idioms, and that might have undergone rejuvenation efforts to remain active until 2023.

In addition, according to (Avelino et al., 2019), inactive repositories are less likely to show code evolutions efforts due to the low frequency of updates after the main developers have abandoned them. The study highlights that even popular projects face difficulties in attracting new contributors, with only 41% of projects fully recovering their maintenance activity after the departure of their main developers. This makes it more challenging to identify sustained rejuvenation efforts in abandoned projects. During data collection, we focused on active repositories to ensure that our analysis captures projects likely to be addressing technological debt and aligning with current best practices.

We used the SEART tool (Dabic et al., 2021) to assist in selecting candidate GitHub repositories, initially identifying 726 JavaScript repositories (S1). In the second step, we aimed to filter out non-relevant repositories. Starting with S1, we applied specific criteria: excluding repositories with fewer than 1,000 commits, fewer than 10 contributors, and any forks or repositories labeled as tutorials. This filtering resulted in a refined set of 286 repositories (S2).

In the third step, we computed the number of JavaScript files for each repository in S2 using the CLOC tool (Danial, 2021). We then calculated the quartiles (Q1, Q2, Q3) based on the number of JavaScript files. We established lower limits for the number of files to filter out repositories with fewer files than these thresholds. As a result, we excluded small projects, that is, those with fewer JavaScript files than the 25th percentile (Q1). For example, repositories that are not actual software products, like the JavaScript style guide found at https://github.com/airbnb/javascript, were also removed from our analysis.

Additionally, 58 repositories were removed due to issues encountered during data collection. Our final selection consists of 158 JavaScript repositories.

3.2 Research organization

We structured our research into two studies. The first one focused on using descriptive statistics and time series to investigate the adoption (and adoption trends) of modern JavaScript features. The first study addresses the research questions RQ1, RQ2, and RQ3; it primarily relies on mining the source code history of the selected repositories. The second study uses thematic analysis (Clarke and Braun, 2017) to highlight the motivations and challenges associated with JavaScript code rejuvenation. We collected information from code review comments during the second study and addressed our last research question (RQ4).

4 First Study: Prevalence and Adoption Trends of Modern JavaScript Features

4.1 Data Collection and Analysis

We examined the source code history of 158 GitHub legacy projects in our dataset to comprehend the prevalence and adoption trends of modern JavaScript features. We developed our own tool (JSMiner) to traverse the source code history of the selected projects and collect usage scenarios for the features we are interested in.

JSMiner relies on an existing JavaScript ANTLR grammar(Parr, 2013). ANTLR is a parser generator widely used for processing structured text or binary data. We tailored an existing ANTLR grammar that specifies the JavaScript concrete syntax. Using the ANTLR tooling, we generate an abstract syntax tree (AST), a parser, and code for traversing the JavaScript AST using the visitor design pattern. This infrastructure enables the analysis of JavaScript programs by parsing and extracting information on specific features and syntactic elements from the source code. Figure 3 presents the data collection pipeline that we use in our study.

Our pipeline uses JSMiner to analyze the evolution of JavaScript projects from 2012-01-01 to 2023-12-31. We begin by generating a list of JavaScript repositories from GitHub using SEART (Dabic et al., 2021). Then, our pipeline uses a Python script to clone all the selected repositories into a local directory.

For each project, JSMiner iterates through the repository's commit history, capturing revisions at 30-day intervals to reduce the total number of analyzed commits. For each selected revision, JSMiner performs a git checkout to retrieve the repository state for the specific commit hash. It then traverses the JavaScript files to extract metrics related to the usage of modern JavaScript features.

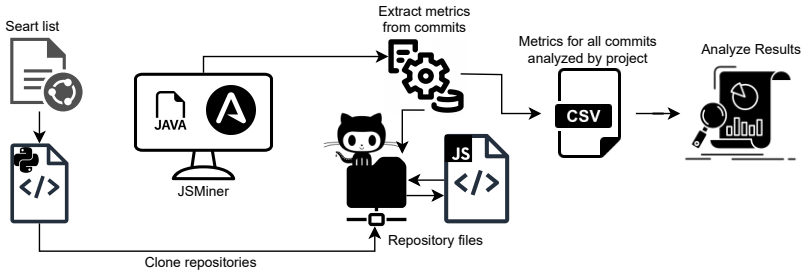

Fig. 3 JSMiner Pipeline for Collecting JavaScript Evolution Data.

We store the collected results for each project in individual CSV files, which contain the extracted data and their corresponding revision identifiers. Finally, we consolidate the individual CSV files into one, enabling the analysis of the results through Python scripts. This workflow ensures a reproducible approach to understanding the adoption of modern JavaScript features across multiple projects over time.

We use this dataset to answer our research questions RQ1, RQ2, and RQ3. To answer the first, we carried out a descriptive analysis that considers (a) the total usage of features in each repository revision and (b) the median number of feature occurrences to estimate the central tendency across the projects. For the second research question, we identified the first occurrence of all modern JavaScript features in the repositories. For the third research question, we modeled feature adoption using a time series model, assigning time and files with feature usage occurrences as variables. We conducted the data analysis using Python scripts.

4.2 Results of Descriptive Analysis

This section presents the results of a quantitative assessment of modern JavaScript feature adoption. We obtained this data by traversing each project's source code history and counting occurrences of the analyzed features. This process allowed us to extract metrics on feature prevalence across projects and determine the first appearance of modern features in our dataset. The results are summarized in Tables 2 and 3, as well as Figure 4, which shows the median percentage of files using each modern JavaScript feature across projects.

The most frequently adopted features in our dataset are Const Declarations, Async Declarations, Arrow Function Declarations, Let Declarations, Enhanced Property Assignment, and Template String Expressions, which are commonly used by more than 75% of projects in our dataset.

Other features that also appear in several projects within our dataset are Object Destructuring, Spread Arguments, Await Operators, For of Statments, Default Parameters, Class Declarations, Function Property Declarations, and

Table 2 Summary of JavaScript Features: Total Occurrences, Adoption Rate in Projects, First Occurrence, and Official Release Year.

Feature	Total of Occurrences (#)	Projects Adoption (%)	First Occurrence	Official Release Year
Const Declarations	218542	91.13	2012-01	ES6 (2015)
Async Declarations	22905	89.24	2012-01	ES2017 (2017)
Arrow Function Declarations	109874	87.97	2014-04	ES6 (2015)
Let Declarations	63274	82.27	2012-01	ES6 (2015)
Enhanced Property Assignment	40697	79.11	2012-01	ES6 (2015)
Template String Expressions	26778	77.21	2015-06	ES6 (2015)
Object Destructuring	8436	68.98	2012-01	ES6 (2015)
Await Declarations	46373	60.75	2015-09	ES2017 (2017)
Spread Arguments	3458	60.12	2015-07	ES6 (2015)
For of Statments	2923	58.22	2014-04	ES6 (2015)
Default Parameters	5102	56.32	2014-05	ES6 (2015)
Class Declarations	5564	53.79	2014-09	ES6 (2015)
Function Property Declarations	7062	51.89	2015-06	ES6 (2015)
Import Statements	48427	51.89	2015-01	ES6 (2015)
Array Destructuring	1811	46.20	2012-01	ES6 (2015)
Export Declarations	18005	43.67	2015-01	ES6 (2015)
Rest Statements	931	37.34	2015-02	ES6 (2015)
Computed Property Assignment	6782	33.54	2015-07	ES6 (2015)
Optional Chain	2909	29.74	2018-05	ES2020 (2020)
Spread in Objects	522	29.11	2015-08	ES2018 (2018)
Null Coalesce Operators	514	15.18	2020-02	ES2020 (2020)
Optional Catch Biding	78	13.29	2019-04	ES2019 (2019)
Yield Declarations	319	10.75	2012-01	ES6 (2015)
Rest in Objects	52	6.96	2016-01	ES2018 (2018)
Private Fields	1004	5.69	2017-07	ES2022 (2022)
Assignment Operators	149	3.79	2020-10	ES2021 (2021)
BigInt	67	3.79	2020-03	ES2020 (2020)
For Await Of Statments	16	3.79	2018-10	ES2018 (2018)
Numeric Separator	32	3.79	2020-09	ES2021 (2021)
Private Methods	657	3.16	2021-11	ES2022 (2022)
Static block in Classes	5	0.63	2023-06	ES2022 (2022)

Import Statements, whose prevalence ranges from 51 to 68% across projects. We also found evidence of less frequent feature adoption of Array Destructuring, Export Declarations, Rest Statements, Computed Property Assignment, Optional Chaining, and Spread in Objects ranging from 29 to 46% across the projects.

Still, eleven features do not appear in many projects in our dataset Null Coalesce Operators, Yield Operators, Private Fields, Numeric Separator, BigInt, Optional Catch Biding, Assignment Operators, Rest in Objects, Private Methods, For Await Of Statments, and Static block in Classes. These features are used by less than 15% of the analyzed projects, suggesting a slight interest in their usage. In particular, our dataset shows no evidence of using the feature Exponentiation Assignments.

Table 3 presents summary statistics for the occurrences of various JavaScript features in GitHub projects. Each feature is listed with its median, mean, standard deviation (std), maximum (max), and minimum (min) number of occurrences across the analyzed projects. JavaScript feature usage varies significantly across projects. For instance, the median usage of Arrow Function Declarations is 121 occurrences, while Const Declarations has a median of 195 occurrences. Other features, such as Async Declarations and Let Declarations, have a median of 23 and 62 occurrences, respectively. These variations highlight differing adoption patterns and usage of JavaScript features within our dataset. In particular, the Handsontable project alone accounts for 18569 occurrences of Arrow Function Declarations, which represents 16.9% of the total, and 21434 occurrences of Const Declarations, making up 9.8% of the total. We

also observe a similar lack of uniformity in the usage of Let Declarations across projects.

Table 3 Summary statistics for the occurrence of JavaScript features in GitHub projects

feature	median	mean	std	max	min
Array Destructuring	0	11.46	38.61	340	0
Arrow Function Declarations	121	695.40	1949.18	18569	0
Assignment Operators	0	0.94	7.79	82	0
Async Declarations	23	144.96	392.73	2861	0
Await Declarations	8.5	293.5	1022.1	8443	0
BigInt	0	0.42	3.12	28	0
Class Declarations	1	35.21	119.94	878	0
Computed Property Assignment	0	42.92	447.34	5621	0
Const Declarations	195	1383.18	3224.63	21434	0
Default Parameters	1	32.29	110.89	1159	0
Enhanced Property Assignment	33	257.57	539.81	2953	0
Exponentiation Assignments	0	0	0	0	0
Export Declarations	0	113.95	301.40	2015	0
For Await Of Statments	0	0.10	0.65	6	0
For of Statments	2	18.5	54.65	472	0
Function Property Declarations	1	44.69	149.04	1136	0
Import Statements	4	306.5	737.74	5319	0
Let Declarations	62.5	400.46	824.66	4347	0
Null Coalesce Operators	0	3.25	15.36	137	0
Numeric Separator	0	0.20	1.67	20	0
Object Destructuring	4	53.39	128.28	888	0
Optional Catch Biding	0	0.49	2.77	33	0
Optional Chain	0	18.41	71.23	513	0
Private Fields	0	6.35	42.87	412	0
Private Methods	0	4.15	28.39	255	0
Rest in Objects	0	0.32	2.11	20	0
Rest Statements	0	5.89	28.56	340	0
Spread Arguments	1	21.88	62.74	551	0
Spread in Objects	0	3.30	9.98	69	0
Static block in Classes	0	0.03	0.39	5	0
Template String Expressions	20	169.48	428.71	3478	0
Yield Declarations	0	2.018	17.20	212	0

Figure 4 shows the median percentage of files across all projects that adopt modern JavaScript features in our dataset. Considering the latest version of the projects, the median percentage of files that use Const Declarations and Arrow Function Declarations is 20% and 17.39%, respectively. This suggests that these features are commonly adopted in the projects and distributed throughout a significant number of JavaScript modules. Using the median instead of the mean avoids the influence of outliers, ensuring a more reliable representation of typical usage.

Let Declarations appear in 9.80% of the files, suggesting its relevance to solving recurrent concerns within the projects and an increasing preference of software developers to adopt modern variable declarations with the modifiers `let` and `const`. Even modern concurrency features like Enhanced Property Assignment (6.25% of the files), Template String Expressions (5.84% of the files), and Async Declarations (4.08% of the files) show recurrent adoption across project files. Features such as Object Destructuring (1.70% of the files) and

(1.08% Await Operators of the files) appear less frequently across JavaScript project files, indicating a smaller prevalence of classical object-oriented patterns. Some features seem to address less recurrent concerns, such as Spread Arguments, For of Statments, Class Declarations, Import Statements, Default Parameters, and Function Property Declarations, which appear in less than 1% of the projects files. This suggests that developers might either be unfamiliar with them or use these features for highly specific problems.

For RQ1, our findings show that modern JavaScript features are used unevenly across projects. Features like Const Declarations, Arrow Function Declarations, Let Declarations, Async Declarations, Enhanced Property Assignment, and Template String Expressions appear in over 75% of projects, with Const Declarations and Arrow Function Declarations found in 20% and 17.39% of files, respectively, whereas features such as Object Destructuring, Await Operators, and Computed Property Assignment appear in less than 5% of files, and Class Declarations, Spread Arguments, and Import Statements in under 1%.

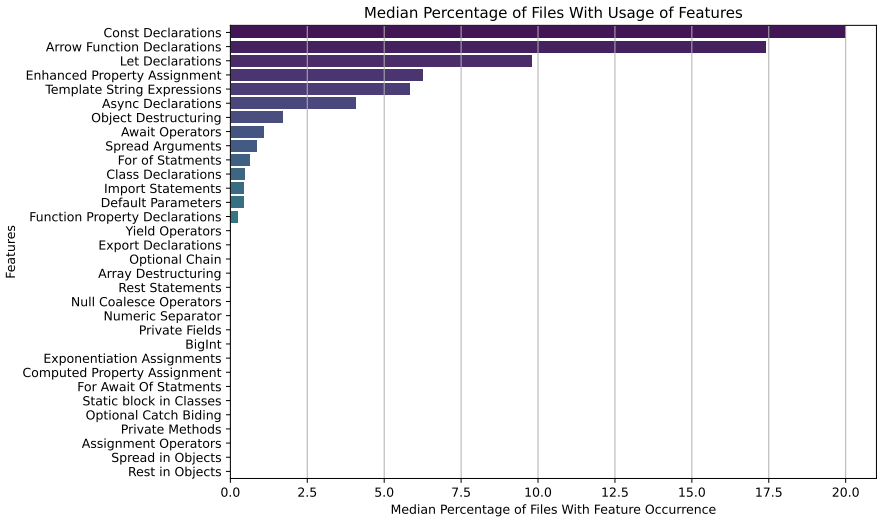

Fig. 4 Modern JavaScript language feature usage by percent of files within projects.

Our dataset reveals that some modern JavaScript features were adopted well before their official language introduction in specific ECMAScript versions, as shown in Table 2. Early-adopted features include Const Declarations, Async Declarations, Let Declarations, Object Destructuring, Array Destructuring, Yield Operators, and Enhanced Property Assignment, all of which began seeing use in January 2012. These features were officially introduced with

ES6 in 2015 and ES2017 in 2017. Features like Arrow Function Declarations, For of Statments, Default Parameters, and Class Declarations saw adoption in April 2014, May 2014, and September 2014, respectively, also preceding the ES6 release. Additionally, Await Operators, Private Fields, Spread in Objects, Rest in Objects, and Optional Chaining saw adoption in September 2015, July 2017, August 2015, January 2016, and May 2018, preceding their release with ES2017 in 2017, ES2022 in 2022, and ES2020 in 2020, respectively.

This early adoption pattern indicates a proactive approach by developers to integrate emerging JavaScript features before their formal standardization. The same was also observed for Java Dyer et al. (2014a). The majority of these features were supported by browsers before the release of ES6 and ES2018. For instance, on June 25, 2013, Arrow Function Declarations was released on Firefox and Firefox for Android [4]. Other features, such as Const Declarations, were added to Safari in 2011 and Opera in 2006 [5].

In contrast, some features were adopted around their official release dates or shortly thereafter. These include Import Statements, Export Declarations, Rest Statements, Template String Expressions, Function Property Declarations, Spread Arguments, Computed Property Assignment, Null Coalesce Operators, BigInt, Numeric Separator, For Await Of Statments, Assignment Operators, Private Methods, Static block in Classes and Optional Catch Biding.

For *RQ2*, the data indicates early adoption by developers. For instance, Const Declarations, Async Declarations, Let Declarations, and Object Destructuring were used as early as 2012, before their official release in ES6 and ES8. Likewise, Arrow Function Declarations and Class Declarations were adopted before the ES6 release in 2015. This pattern suggests that developers actively explore and incorporate emerging features as browser support becomes available, demonstrating a proactive approach to modernization.

4.3 Trends of features adoption

In this section, we analyze the adoption trends of various modern features in a software development context. We employed statistical techniques to evaluate the stationarity and trend direction of time series data associated with each feature. The dataset comprises monthly counts of unique files containing occurrences of feature usage, extracted from software development repositories. The features analyzed include Const Declarations, Async Declarations, Arrow Function Declarations, Let Declarations, Enhanced Property Assignment, Template String Expressions, Object Destructuring, Await Operators,

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#browser_compatibility

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#browser_compatibility

Spread Arguments, For of Statments, Default Parameters, Class Declarations, Function Property Declarations, Import Statements, Array Destructuring, Export Declarations, Rest Statements, Computed Property Assignment, Optional Chaining, Spread in Objects, Null Coalesce Operators, Optional Catch Biding, and Yield Operators, which are found in at least 10% of the programs analyzed in our dataset, as shown in Table 2 in Section 4.2.

We performed an Augmented Dickey-Fuller (ADF) test to assess stationarity (Vishwas and PATEL, 2020; Huang and Petukhina, 2022). If the p-value of the ADF test was less than or equal to 0.05, the series was deemed stationary, indicating a rejection of the null hypothesis. Conversely, if the p-value exceeded 0.05, the series was considered non-stationary.

Our analysis of modern JavaScript features, using the Augmented Dickey-Fuller (ADF) test and the statistical smoothing method of Loess (Locally Weighted Scatterplot Smoothing) (Cleveland, 1979), aims to assess if there is a trend in the time series data of each feature's usage. The ADF test reveals that all features exhibit non-stationary behavior, with p-values significantly above the 0.05 threshold, meaning that the null hypothesis of a unit root cannot be rejected. This non-stationarity suggests that the usage patterns of these features do not revert to a mean or trend-stationary process but continue evolving. We then applied Loess smoothing to better visualize the trends, which helped uncover points of significant change in the usage trajectory of each feature. Our findings show that the adoption trends of modern JavaScript features generally align with, or follow, their introduction in the language, though some features experienced earlier uptake.

For instance, Const Declarations, which began its uptake in 2017, closely followed its introduction in ES6 (2015). ES6 introduced modern features like Let Declarations, Object Destructuring, Arrow Function Declarations, Default Parameters, For of Statments, Spread in Objects and Template String Expressions, which developers gradually integrated into their coding practices between 2016 and 2017 after the initial release. Similarly, features such as Class Declarations, Import Statements, and Export Declarations saw increased adoption starting around 2017, aligning with their inclusion in ES6. Figure 5 present the time series for Const Declarations and Figure 6 present the data from Let Declarations, Object Destructuring, Arrow Function Declarations, Default Parameters, and Template String Expressions.

Features like Rest Statements and Function Property Declarations began gaining traction earlier, dating back to 2015, reflecting developers' early recognition and utilization of these modern features shortly after their introduction in ES6. In contrast, features introduced in subsequent ECMAScript versions, such as Await Operators (ES2017, 2017) and Optional Chaining and Null Coalesce Operators (ES2020, 2020), show a delay in initial growth adoption compared to their release dates, with noticeable upticks in usage appearing around 2020 and 2021, respectively.

This trend might underscore a typical pattern in programming language adoption. Developers often wait for features to stabilize, tooling support to mature, and community best practices to emerge before incorporating new

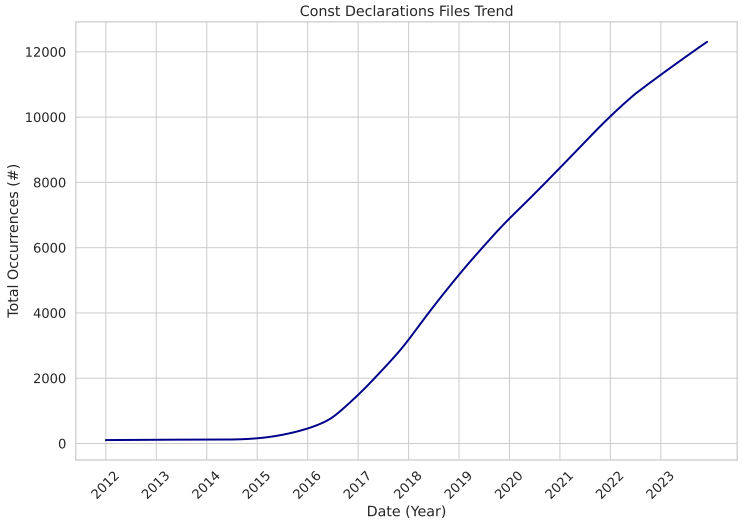

Fig. 5 Initial growth in the adoption of Const Declarations in 2017 after the release in ES6 (2015).

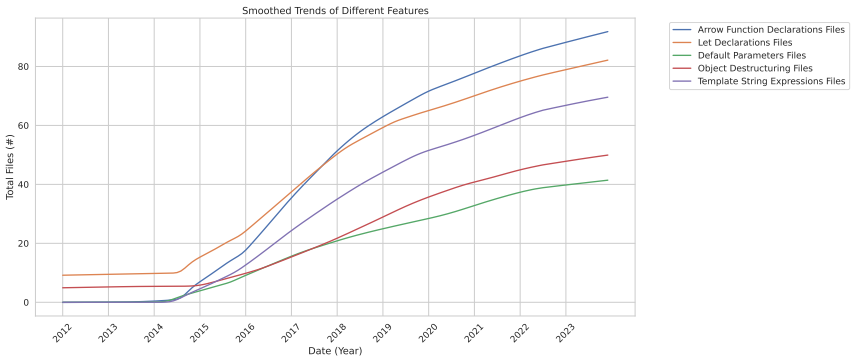

Fig. 6 Initial growth in the adoption of Let Declarations, Object Destructuring, Arrow Function Declarations, Default Parameters, and Template String Expressions between 2016 and 2017 after the release in ES6 (2015).

language constructs extensively into their projects. A study of KDE developers (Lucas et al., 2023) reveals a similar pattern: although developers started using *auto-typed declarations, lambda expressions, and range-based for loops* before 2011, significant growth in their usage was observed only after 2016, five years after the release of the C++11 specification. This delay highlights the necessity for a maturation period before growth adoption can occur. This suggests a broader trend where the adoption of new language features is not immediate. Instead, it requires time for the development ecosystem to fully integrate modern features, during which developers adapt to new paradigms,

update their codebases, and establish new best practices. Consequently, the pattern seen in JavaScript's adoption of modern features like Const Declarations and Arrow Function Declarations aligns with this broader phenomenon observed across different programming languages and development communities.

For *RQ3*, the adoption of modern features evolves over time. Some features were quickly embraced after their ES6 release, while others, such as Await Operators and Optional Chaining, only increased in use after 2020, suggesting a gradual integration as tooling and best practices mature.

5 Second Study: Motivations and Challenges for Rejuvenating JavaScript Code

5.1 Data Collection and Analysis

To understand the motivations and challenges that developers face when rejuvenating JavaScript code, and to answer our fourth research question, we manually analyzed a subset of code review comments from the repositories analyzed in our first study. In the scope of this research, a code review consists of evaluating merge requests (also known as pull requests) to identify potential issues with the code, such as defects, security vulnerabilities, and poor design choices (Oliveira, 2021; Bogachenkova et al., 2022; Lin and Thongtanunam, 2023).

Currently, this is a common practice in collaborative development, where a developer submits changes to the central repository and asks other developers to review those changes, providing feedback and improvement recommendations. The interaction between contributors (developers proposing changes) and reviewers continues until the changes are considered ready to be integrated. Similar to other software development platforms (such as GitLab and BitBucket), GitHub facilitates the code review process through pull requests, where reviewers can attach general feedback by commenting on specific parts of the modified source code. These comments are important for highlighting specific and fine-grained concerns of reviewers.

We developed a Python infrastructure that leverages the GitHub API to scrape comments from pull requests, automating the data collection process. This infrastructure iterates over each pull request in the repositories in our dataset and retrieves the related comments, extracting relevant information (such as user login names, URLs, timestamps, and message bodies) from the response data. Subsequently, it locates the corresponding repository in the

database and stores the pull request comments, associating them with the respective pull request IDs.

In total, we mined 122,248 pull request comments and populated a database for analysis. We then executed a textual query to identify potential pull request comments related to source code rejuvenation. This query searched for mentions of modern JavaScript features (e.g., *"arrow function"*, *"let declaration"*, *"const declaration"*) and specific ECMAScript versions (e.g., *"ES6"*, *"ECMAScript 2015"*). Additionally, the query included variations and related terms, such as *"arrow function implementations"*, *"async declarations"*, *"template string usage"*, and *"numeric separator inclusion"* to capture a broader range of relevant discussions. This approach resulted in 447 pull request comments, which were manually analyzed to determine their relevance to rejuvenation efforts and feature adoption.

Regarding data analysis, we employ a thematic analysis (Clarke and Braun, 2017) approach to extract **motivations** and **challenges** for rejuvenating JavaScript code from code review comments. We leverage Saldaña's recommendations for data coding (Saldana, 2012). Accordingly, we start with open coding, which refers to the initial step of analyzing qualitative data by selecting and associating relevant segments of text to generate labels or tags that directly correspond to concepts and categories derived from the data. We then proceed to classification, wherein we systematically arrange the original codes into distinct categories based on their shared qualities. During this process, we consistently improved and reevaluated codes and categories as new information and insights emerged.

This data analysis procedure ran in two cycles to ensure the relevance and accuracy of the selected pull request comments. In the first cycle, one author of this paper reviewed the 447 code review comments identified in the textual query, separating those relevant to rejuvenation efforts from those merely discussing the use of a modern JavaScript feature. From this process, 93 comments were selected as potentially relevant to rejuvenation. In the second cycle, three authors of this paper independently evaluated a subset of 31 comments. They independently analyzed the comment and the surrounding pull request discussion to verify whether it represented efforts to rejuvenate existing code to adopt modern features. After coding these comments independently, the other two authors reviewed each other's work, indicating agreement or disagreement with the coding. For comments where disagreements arose, all three authors discussed the coding until a consensus was reached.

Finally, the first author of this paper categorized the codes into thematic groups, resulting in three primary categories for JavaScript source code rejuvenation: Motivations (Section 5.2), Challenges (Section 5.3), and Transpiler-based approach (Section 5.4). These categories are detailed in the remainder of this section. Furthermore, during the assessment, we identified several contributions to the repositories that significantly rejuvenate legacy JavaScript code (e.g., one contribution replaces dozens of old-style constructs and idioms with arrow functions). In Section 5.5, we discuss some of these *"rejuvenation efforts"*.

5.2 Motivations to source code rejuvenation

This category presents the motivations that lead JavaScript developers to make efforts to rejuvenate the source code of their applications—that is, replace legacy constructs and idioms with new features of a programming language. According to our findings, ***improving code comprehension*** is the most frequently mentioned by developers during the code review process in our study that leads JavaScript developers to rejuvenate legacy source code (with 10 occurrences). Code comprehension is the process through which developers actively acquire knowledge about a software system by exploring and studying various software artifacts. This includes reading and understanding the source code and related documentation (Daka et al., 2015). Different coding aspects, such as programming constructs, naming conventions, and formatting, influence how easily developers can comprehend code (Oliveira et al., 2020). We considered here aspects like *"conciseness"* and *"clearness"* as code comprehension.

For instance, in the code review comment of Figure 7, a developer suggests that using an Arrow Function Declarations would make the code more concise and elegant. The motivation behind the comment suggestion is to encourage the adoption of ES6 features, specifically Arrow Function Declarations, to make the code more concise. By replacing the traditional `anonymous function` with an Arrow Function Declarations, the Arrow Function Declarations syntax simplifies the code by eliminating the need for the `function` keyword and reducing the overall verbosity of the code. According to (Rauschmayer, 2015), one of Arrow Function Declarations goals is to shorten syntactical form.

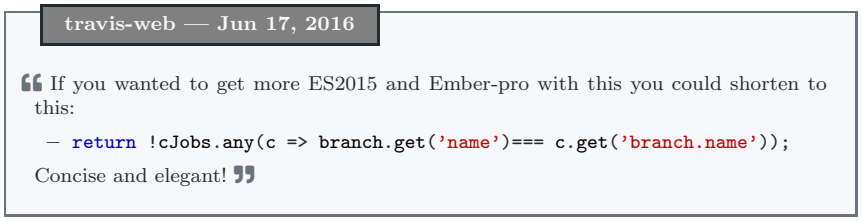

Fig. 7 Suggestion to adopt Arrow Function Declarations to improve code comprehension.

Another benefit of the adoption of Arrow Function Declarations that improves code comprehension is that it binds the `this` keyword to the surrounding lexical context. According to (Rauschmayer, 2015), the developers can eliminate the need for the statement `self = this` to preserve the correct `this` context within nested functions. This happens because the use of Arrow Function Declarations eliminates the need for manual binding of `this`, ensuring that the correct context is preserved. This makes the code more concise and reduces the risk of errors due to incorrect `this` context. Also, developers recommend adopting Async Declarations, Await Operators, and Object Destructuring to make the code more clear and concise.

One developer suggested a source code rejuvenation (in Figure 8) to use asynchronous programming and object destructuring. By employing Async Declarations and Await Operators, the code now handles the promise returned by readArmored more cleanly, ensuring that this block of code is not executed before the promise is resolved or rejected. Additionally, Object Destructuring simplifies the extraction of the first key from the keys array, resulting in more concise code. According to the proposal by tc39 [6], one of the goals of async and await features is to reduce the promise's boilerplate code and make syntax concise.

```
it('require_uid_self_cert=true: cannot use pubkey w/o self certs', async
     function() {
- let k = openpgp.key.readArmored(no_self_cert one_user_no_self_cert).keys[0];
+ let {keys: [k]} = await
     openpgp.key.readArmored(no_self_cert.one_user_no_self_cert);
  ...
}
```

Fig. 8 Async/await and destructuring for cleaner promise handling and key extraction from project *openpgpjs* in the pull-request #753.

5.3 Challenges to source code rejuvenation

This category covers the challenges that hinder JavaScript developers from rejuvenating the source code of their applications. ***Incompatibility*** is the most frequent challenge in our assessment, appearing a total of 25 times. For instance, developers emphasize possible incompatibility issues between modern JavaScript features as older browsers and dependencies, which often prevent them from using modern JavaScript. In Section 2.4 we presented a brief discussion about about JavaScript Cross-Version Compatibility.

In the code review comments in Figures 9 and 10, developers discuss incompatibility issues between modern JavaScript and browsers like PaleMoon, Chrome versions prior to 41, IE11, and older versions of Safari.

adsbypasser — May 6, 2017

> ES6 is good but not all target platforms support `let`. (e.g.: PaleMoon or Chrome ¡ 41)

Fig. 9 Avoiding source code rejuvenation due to incompatibility issues with old browsers.

[6] https://tc39.es/proposal-async-await/#intro

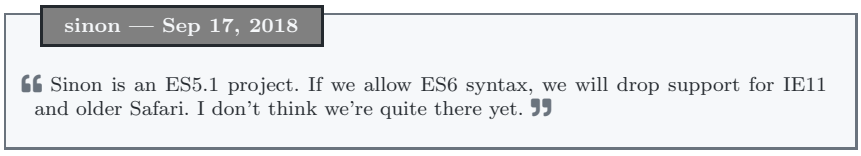

Fig. 10 Avoiding source code rejuvenation due to incompatibility issues with old browsers.

We also found code review comments where developers report compatibility issues between modern features and JavaScript libraries, platforms, and tools—such as UglifyJS and PhantomJS. See code review comments in Figures 11 and 12. UglifyJS [7] is a JavaScript tool used to minify and compress JavaScript code in order to improve processing time by removing unnecessary characters, optimizing code structure, and reducing file size. PhantomJS [8] is a headless web browser, meaning it operates without a graphical interface. It is used primarily for automating web page interactions, testing, and scripting web pages, allowing developers to run and test their code in a controlled environment.

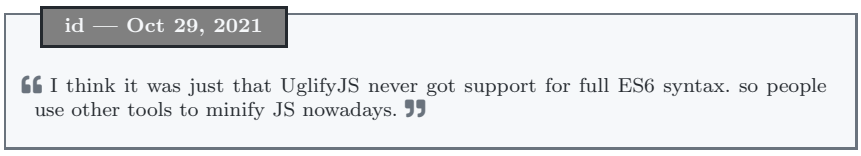

Fig. 11 Avoiding source code rejuvenation due to incompatibility issues with dependencies.

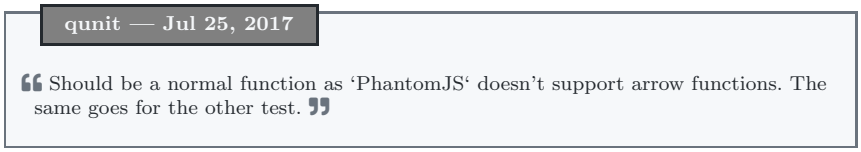

Fig. 12 Avoiding source code rejuvenation due to incompatibility issues with platforms.

Also, developers report incompatibility issues between modern JavaScript and runtime engines (see the code review in Figure 13). For instance, versions of Node.js up to and including v0.12, as well as the JavaScript Nashorn [9], do not support modern JavaScript. Nashorn was embedded in versions 8 – 15 of the Java Development Kit. Legacy versions of these engines provide limited or nonexistent support for the new features introduced in ES6. Nonetheless, the

[7] UglifyJS: https://www.npmjs.com/package/uglify-js
[8] PhantomJS: https://github.com/ariya/phantomjs
[9] JEP 372: https://openjdk.org/jeps/372

projects we investigate often need to maintain compatibility with these legacy versions.

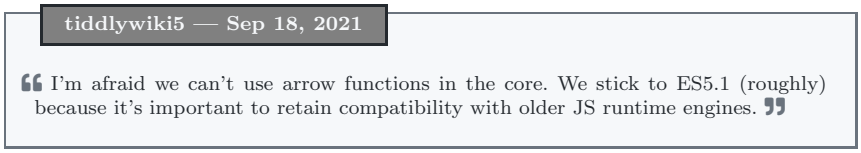

Fig. 13 Avoiding source code rejuvenation due to incompatibility issues with JS runtime engines.

Code conventions might prevent the adoption of modern JS features is the second most common challenge in our assessment, with 4 occurrences. In this case, developers report that they prefer to maintain consistency with existing code conventions rather than adopt modern features. For instance, the code review comment in Figure 14 shows that a specific developer prefers to avoid using modern JavaScript features to maintain consistency with the legacy code. This resistance to change can hinder the adoption of new features, potentially limiting the benefits of source code rejuvenation.

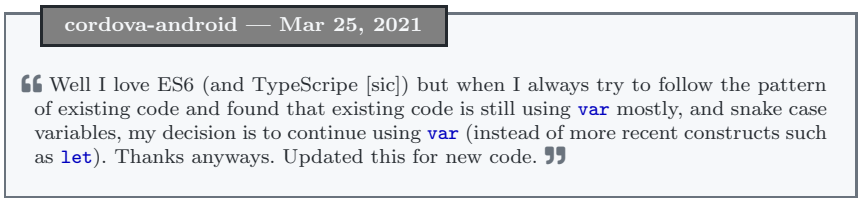

Fig. 14 Developers prefer to maintain consistency with the existing code instead adopt modern features.

Code review comments also suggest that ***modern features might reduce the program's performance***. In the code review comment in Figure 15, a contributor states that newer features like `Map`, Let Declarations, and Const Declarations might compromise system performance. The full discussion in this pull request highlights a performance issue with using modern features in certain contexts, where slower performance was observed compared to "legacy" alternatives. As a result, there's a tradeoff decision about whether to use a `Map` or stick with the legacy approach. The suggestion to add a `TODO` to consider `Map` when the performance improves indicates a willingness to revisit the issue in the future.

Additionally, there's consideration of optimizing code for performance on Long Term Support (LTS) versions of Node.js versus staying with the latest versions, with a leaning towards staying up-to-date and incorporating updates into the build performance guide. This code review underscores the importance

of balancing the benefits of new features with performance considerations when making decisions about source code rejuvenation.

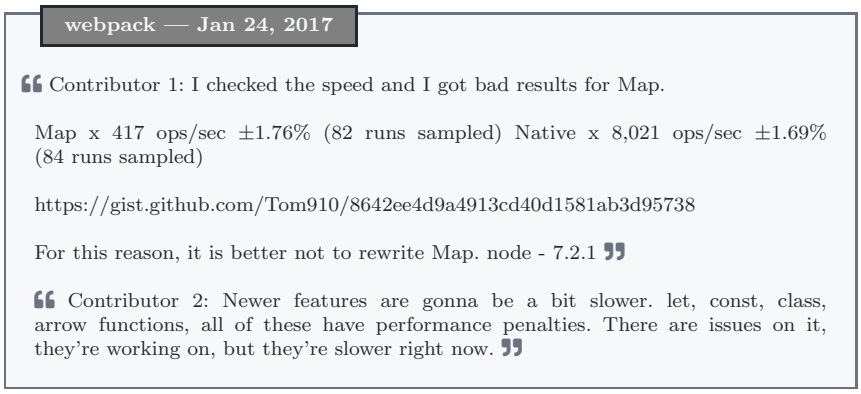

Fig. 15 Modern features might reduce the program performance.

We also found recommendations to ***avoid using features that were recently introduced***. Our analysis and the previous results outlined in this section suggest that the reluctance to adopt recent features (such as Optional Chaining) may stem from its novelty and the lack of full support across all execution environments. Therefore, developers might prefer to avoid modern features like Optional Chaining and opt for more traditional methods of verifying nested properties to ensure compatibility with various execution environments. Code review comments also suggest that modern features might be more complex for less experienced JavaScript developers. Introducing new syntactic constructs, such as Arrow Function Declarations, could potentially raise the learning curve, making it harder to understand and work with the code, especially for new contributors or those who don't have JavaScript as their primary programming language. Although the reviewer comment in Figure 16 acknowledges the benefits of Arrow Function Declarations, it highlights that these benefits do not outweigh the disadvantages and additional complexity introduced. The same comment highlights that development tools with built-in shortcuts can prevent developers from adopting new syntax simply because these tools make it easier to write code using the old syntax (see Figure 16). For instance, an editor shortcut that quickly generates an empty function template might discourage the use of Arrow Function Declarations, even though the latter would make the code more concise. This reliance on shortcuts for older syntax can hinder efforts to rejuvenate the codebase, as developers might default to using the more familiar and easily accessible patterns provided by their tools.

video.js — Apr 11, 2015

> I don't know how I missed these but they're the same as the other example we reverted. I looked at how babel translates arrow functions and it at least doesn't do an extra 'fn.bind()' so that's good. I want to make sure I'm not just being a luddite here so if there's fault in my reasoning let me know. I think it's an important issue. I still don't think the shortened syntax alone is benefit enough to warrant upping the javascript reading level for contributors. For most contributors JavaScript is probably still not their primary programming lang and we'd be replacing one of the most most well known keywords with one of the least known that does the exact same thing (in this specific case). Also Babel makes named functions when translating arrow functions. That goes against the "Arrow functions are always anonymous" rule and could potentially [create issues in IE8](http://kiro.me/blog/nfe_dilemma.html). Though I'm not sure how likely that is and I wish we could use named functions more for stack traces. Also if you have a text editor with a function shortcut there's not much benefit left over for this specific case. 'f' then 'tab' quickly in Atom makes an empty function. I do think there are some cool examples of using arrow functions like: (...) I would have a harder time arguing against that and it even seems more readable than the example above for some reason. And using it for it's 'this' binding. (...) But in all the examples here we seem to just be using it for it's shorter anonymous function syntax which if that's correct then for consistency there's over 500 other anonymous functions in the codebase we would need to update too. Am I missing another benefit in these examples?

Fig. 16 Examples of challenges in adopting modern features. First the comment illustrates the reluctance to use Arrow Function Declarations due to the potential increased complexity for contributors unfamiliar with JavaScript. Second, demonstrates how development tools with built-in shortcuts for older syntax can discourage the use of newer, more concise syntax, thus hindering source code rejuvenation efforts.

5.4 Transpiler-based approach to rejuvenate JavaScript code

This category summarizes some of the code review assessment findings related to ***the adoption of a transpiler (e.g., Babel) to support developers*** in using modern JavaScript features in legacy code. Indeed, we found code review comments suggesting that developers transpile modern JavaScript features using Babel (see Figure 17). Transpiling modern JavaScript code can increase the compatibility of the code with legacy JS engines, addressing challenges related to their differing support for new syntax features. As such, developers can write code using the latest syntaxes, and transpile it to widely supported legacy constructs and idioms. This approach allows for the immediate use of advanced features without waiting for general support in JS engine implementations, facilitating the source code rejuvenation process.

However, as developers noted, the use of a transpiler might introduce complexity in the build process and potential confusion in code organization, particularly when mixing properties in constructors with other initialization code. Another side effect is that ***using a transpiler might introduce bugs***. The code review comment in Figure 16 highlights that Babel's transpilation of Arrow Function Declarations into named functions can cause compatibility issues with older browsers. Additionally, issues have been reported in the Babel repository regarding incorrect transpilation of arrow functions, as seen

in GitHub issues Issue #1742, Issue #1453, and Issue #1887. These issues illustrate how the use of transpilers can inadvertently introduce new bugs, complicating efforts to rejuvenate the codebase.

We also found code review comments reporting that ***the Babel transpiler might significantly increase the overhead of the resulting code***. In the comment in Figure 18, a code reviewer explains that they used ES5 features outside the modules because of the significant overhead added by Babel when transpiling from ES6 to ES5. Even if only a small number of ES6 features were used, the resulting transpiled code might degrade the performance of the scripts. This is due to the additional code Babel includes to ensure compatibility with older browsers, leading to a much larger final bundle that potentially impact performance and load times.

novnc — Jan 30, 2018

> Contributor: Yup. Declaring variables as touchButton = 1 as you can do with methods is ES7 so it would need transpilation to all browsers. So the way to declare value properties in ES6 is to declare them like this in the constructor. I've put them always at the bottom so it should be trivial to take them out once ES7 is more widespread.
>
> Member: Urgh. Are there any other patterns that can be used? I'd like to avoid the confusion of putting it in the constructor with other stuff. And for RFB it gets extra messy as it has getters and setters for some properties but not all.
>
> Contributor: I think that we could use https://babeljs.io/docs/plugins/transform-class-properties/ However we would need to transpile it for all browsers AFAIK. It's not even part of es2017 preset.
>
> Member: That's a no-go in that case. We need something that everyone (except IE) supports.
>
> Contributor: Agreed. That's why I opted for adding them at the bottom of the constructor with a comment stating that they are properties. Once the class properties become a standard, it's trivial to move them out of there.. . .

Fig. 17 Suggestion to use Babel transpiler to maintain compatibility with old browsers.

pouchdb — Oct 13, 2015

> Yeah I am very new to ES6 modules and I was in a hurry so the format I used was mostly determined by a search-and-replace. Also I had to be careful to only use ES5 features _outside_ the modules because I found that using 'babel' to transpile from ES6 to ES5 added a huuuge amount of overhead even if I was only using one or two features in one or two places.

Fig. 18 Babel's transpilation might increase overhead.

For *RQ4*, our findings indicate that enhancing readability, conciseness, and maintainability drives the adoption of modern JavaScript features. Yet, challenges like compatibility, resistance to change, and performance concerns can delay their uptake. Tools such as Babel help introduce modern syntax in older environments, though they also add complexity and potential risks. This highlights the need for a balanced strategy that aligns modernization efforts with project constraints and ecosystem stability.

5.5 Rejuvenation Efforts

During our code review assessment, we identified 31 commits aimed at rejuvenating JavaScript code (referred to as Rejuvenation Efforts Commits). Although this finding does not directly address any research question, we believe it is still useful to characterize JavaScript rejuvenation efforts in this paper. These 31 commits come from 19 distinct projects in our dataset, representing 12% of the 158 projects. They target 12 modern JavaScript features in total. Table 5 shows the number of features adopted in the projects by source code rejuvenation with corresponding commit. In the rest of this section, we will describe some of the source code rejuvenation efforts.

For instance, a large effort to adopt modern JavaScript in commit (Figure 19) introduces more than **2038 Const Declarations** and **152 Let Declarations** to replace *Var Declarations*. The listing in Figure 20 is also part of a test case from *Jasmine* project, focusing on a scenario where a fallback to `setTimeout` occurs. The rejuvenation replaces the traditional variable declaration `(var)` with the modern Const Declarations keyword for the `setTimeout` function. Additionally, the code updates the assignment of the variables `clearStack` and `called` using the Let Declarations keyword for second variable. These changes are part of a large effort to adopt more modern JavaScript syntax and follow some best practices (Haverbeke, 2014; Cantelon et al., 2013). The functionality of the test case, which checks the fallback behavior to `setTimeout`, remains unchanged.

Commit 1 *(project `jasmine`)*
- *Hash: 1166d10e4*
- *Date: April 16, 2022*
- *Message: Use const/let in specs, not var*

Fig. 19 Rejuvenation effort commit from project `jasmine`

Several projects have adopted Arrow Function Declarations to rejuvenate their codebase. For instance, the commit depicted in Figure 21 introduces **92 Arrow Function Declarations**. This commit is particularly interesting

```
it('falls back to setTimeout', function() {
- var setTimeout = jasmine.createSpy('setTimeout').and.callFake(function(fn) {
- fn();
- }),
+ const setTimeout = jasmine
+ .createSpy('setTimeout')
+ .and.callFake(function(fn) {
+ fn();
+ }),
       global = { setTimeout: setTimeout },
- clearStack = jasmineUnderTest.getClearStack(global),
- called = false;
+ clearStack = jasmineUnderTest.getClearStack(global);
+ let called = false;
...
```

Fig. 20 Example of introducing Const Declarations and Let Declarations in a large commit from `jasmine`.

because it also introduces Await Operators, Async Declarations, and Object Destructuring. This specific change replaces a traditional JavaScript function declaration with the more concise Arrow Function Declarations syntax (Haverbeke, 2014) and revealing the developer interest in rejuvenating test code. Commit in the Figure 22 introduces **111 Arrow Function Declarations** in a big effort to rejuvenate their code.

Commit 2 *(project `highlight.js`)*
- *Hash: a69eb6a2e*
- *Date: May 26, 2019*
- *Message: Switch to fs.promises vs bluebird for tests (#2226) - also switch to more idiomatic JS using the newer syntax*

Fig. 21 Rejuvenation effort commit from project `highlight.js`

Commit 3 *(project `noVNC`)*
- *Hash: 651c23ece*
- *Date: Jul 12, 2018*
- *Message: Use fat Arrow Function Declarations `const foo = () => ... ;` for callbacks and any other function that is passed around and it's not a top level function.*

Fig. 22 Rejuvenation effort commit from project `noVNC`

Finally, commit in Figure 23 introduces the Import Statements syntax in a JavaScript module. Specifically, the transformation replaced the CommonJS

`require` statements with ES6 Import Statements. The transformation from CommonJS to ES6 module syntax in the PouchDB project providing a more concise and clear structure for dependencies. Finally, ES6 modules support tree shaking, which can reduce bundle sizes and improve performance by eliminating dead code. The listing in Figure 24 present one of the transformations in this commit.

Commit 4 *(project `PouchDB`)*
- *Hash: 99c24a58*
- *Date: Jan 1, 2016*
- *Message: Migrate to ES6/Rollup, build one index.js.*

Fig. 23 Rejuvenation effort commit from project `PouchDB`

```
- 'use strict';
-
- var createBlob = require('../../deps/binary/blob');
- var Promise = require('../../deps/promise');
- var idbConstants = require('./constants');
- var DETECT_BLOB_SUPPORT_STORE = idbConstants.DETECT_BLOB_SUPPORT_STORE;
+ import createBlob from '../../deps/binary/blob';
+ import Promise from '../../deps/promise';
+ import { DETECT_BLOB_SUPPORT_STORE } from './constants';
...
```

Fig. 24 Refactoring was to rejuvenate the source code by adopting ES6 module syntax from `pouchdb`.

6 Discussion

In this section, we discuss the implications of our findings (Section 6.1) and the limitations that may threaten the validity of our research (Section 6.2).

6.1 Implications of our Findings

We performed an extensive analysis of 158 JavaScript programs from GitHub. We assessed the adoption of modern JavaScript features, such as Const Declarations, Async Declarations, Arrow Function Declarations, Let Declarations. Additionally, we manually analyzed a subset of code review comments from the repositories used in our dataset. Within this part, we provide a concise overview of the implications of our research on some topics:

The results for our first research question (RQ1) suggests that the adoption of modern JavaScript features varies significantly across open-source projects. Widely used features like Const Declarations, Async Declarations/Await Operators, and Arrow Function Declarations are prevalent, while others like Null Coalesce Operators and BigInt are less commonly integrated. This suggests that developers prioritize features that offer immediate benefits in code quality and maintainability, while more specialized features see selective use.

The findings from RQ2 and RQ3 reveal a nuanced pattern in adopting modern JavaScript features. On the one hand, RQ2 highlights that developers often adopt new features before they are officially standardized, using this early period to test and understand their potential. This proactive approach allows developers to experiment with cutting-edge tools and integrate them into their projects even before the browser's full support is available. However, the broader trends identified in RQ3 suggest that widespread adoption typically follows a more measured approach. Developers tend to wait for features to stabilize, for tooling to mature, and for best practices to emerge before fully embracing these new modern features across their projects. For the effective integration of new language features into development practices, this maturation period is crucial, balancing the benefits of innovation with the reliability required for long-term project success. This pattern is not unique to JavaScript but is also observed in the adoption of new features in other programming languages (Lucas et al., 2023).

For the implications from RQ4, our results suggest that while modern JavaScript features improve code comprehension and maintainability, they also introduce challenges, particularly regarding compatibility with older environments. Tools like Babel help bridge these gaps, but they can also add complexity and potential bugs, requiring developers to balance the benefits of modern features with the practicalities of legacy support. Our findings offer practical examples that can assist software developers in rejuvenation the source code of their programs, along with insights for tool developers to identify opportunities for suggesting the adoption of modern JavaScript features.

6.2 Threats to Validity

A potential threat to the internal validity of our study arises from the influence of automated tools, such as transpilers and refactoring tools, on the observed adoption of modern JavaScript features. For example, the Handsontable project alone accounts for 18,569 occurrences of Arrow Function Declarations (16.9% of the total) and 21,434 occurrences of Const Declarations (9.8% of the total). Such outliers may indicate tool-driven transformations rather than adoption by developer actions. However, our study does not include an in-depth evaluation to distinguish whether the usage of these features resulted from automated rejuvenation efforts or deliberate developer actions. While this limitation prevents us from asserting "manual rejuvenation efforts," this limitation does not compromise the validity of our findings.

For external validity, our analysis involved 158 out of 270849 JavaScript projects (0.05%) sourced from the GitHub community. These selected projects met specific criteria, including a development history of more than ten years and recent contributions (after January 1, 2023). It is important to acknowledge that our dataset represents only a small fraction of the total number of applications, limiting the generalization of our findings. Additionally, since our study focused solely on open-source repositories, we cannot generalize our results to industry projects.

Furthermore, the difference in the distributions of feature adoption based on our current dataset (with extremely small projects removed and without our filtering criteria) reveals an important limitation in generalizing our findings to smaller projects. When considering projects with less than 25% of the JavaScript files, we identified repositories that have lower adoption of certain features compared to larger projects. For example, in table 4, the adoption of Const Declarations was 71.52% among projects, while in the table with removal of smaller projects, this adoption rose to 91.14%. Similarly, adoption of Arrow Function Declarations increased from 67.09% to 87.97%, reflecting the higher prevalence of more modern features in projects with larger amounts of JavaScript files. This phenomenon indicates that the results of feature adoption are not evenly distributed, with an indication that modern features are used in a reduced number of smaller projects. As a consequence, we cannot generalize our conclusions to smaller-scale projects, as the characteristics of larger repositories significantly influence the prevalence of these technologies.

Table 4 Summary of JavaScript Features: Total Occurrences, Adoption Rate in Projects, First Occurrence, and Year of Release without filtering smaller projects.

Feature	Total of Occurrences (#)	Projects Adoption (%)	First Occurrence
Const Declarations	87337	71.51	2012-01
Async Declarations	13288	68.98	2012-01
Arrow Function Declarations	52318	67.08	2013-11
Let Declarations	37484	66.45	2012-01
Enhanced Property Assignment	26266	60.75	2013-08
Template String Expressions	14048	58.86	2014-07
Object Destructuring	5119	46.83	2013-08
Spread Arguments	1224	43	2014-11
Default Parameters	1444	42.40	2014-06
Await Declarations	19560	37.34	2016-02
Import Statements	17581	36.70	2013-07
Function Property Declarations	6508	36.70	2014-11
For of Statments	1398	36.07	2014-07
Class Declarations	1716	35.44	2014-01
Export Declarations	6498	32.91	2013-06
Array Destructuring	556	29.11	2013-08
Computed Property Assignment	714	24.05	2014-11
Rest Statements	287	22.78	2014-05
Spread in Objects	247	15.82	2015-11
Optional Chain	1004	13.29	2020-09
Null Coalesce Operators	185	6.32	2020-09
Rest in Objects	25	5.69	2019-03
Optional Catch Biding	58	3.16	2016-12
Yield Declarations	242	3.16	2012-01
Private Fields	83	2.53	2022-09
For Await Of Statments	8	1.89	2020-11
Private Methods	24	1.26	2022-09
BigInt	1	0.63	2023-01
Numeric Separator	2	0.63	2021-10

However, we believe that our study offers valuable insights into the development practices of JavaScript developers, which can be relevant for similar contexts and provide guidance for developers working on mature and evolving codebases.

Regarding the qualitative investigation, we utilized coding methods to analyze the pull-request comments. Despite efforts to mitigate bias by involving multiple coders, the authors' experience and motivations might have influenced the process. While we aimed to uncover and report the motivations and challenges of source code rejuvenation through code review, some limitations may still exist. We describe these limitations based on the steps taken to enhance confidence and validity. Additionally, we followed approaches explored in previous studies (Bacchelli and Bird, 2013; Sadowski et al., 2018; Oliveira, 2021).

Still, on external validity, another potential threat is related to the temporality of the analyzed code review comments. Although we collected comments from 2012 to 2023, the relevant comments regarding code rejuvenation efforts found in our study were only from the period between 2015 and 2021. During this period, the JavaScript programming language have undergone considerable change and maturation, especially with support for modern features through transpilers such as Babel or syntactic supersets such as TypeScript. As a result, the challenges identified in the study may not fully reflect the current reality, limiting the generalizability of the results in terms of more recent development practices. To overcome this limitation, we recommend that developers and readers should also interpret the study's recommendations as cautions about potential challenges, such as compatibility issues and performance overhead.

7 Related Works

Our study builds upon prior research that explores the adoption of modern programming language features across various languages, including Kotlin, Python, TypeScript, C++, Java, and JavaScript. Specifically, it investigates the motivations, challenges, and trends associated with adopting and utilizing these features. In this context, we categorize the related work into two main areas:

(i) **Tools and Techniques for Code Rejuvenation** – Studies that focus on methodologies, frameworks, and automation strategies for integrating modern language features into existing codebases.
(ii) **Feature Adoption and Usage Analysis** – Research examining how and why developers adopt modern language features, as well as the reasons behind developers' adoption of modern language features and the challenges they face during this process.

7.1 Tools and Techniques for Code Rejuvenation

As programming languages evolve, new features are introduced to enhance code comprehension, improve conciseness, and streamline development. Following these advancements, developers seek strategies and techniques to adapt existing codebases to incorporate these modern features. For example, Kumar et al. examined the rejuvenation of legacy C++ programs by replacing C preprocessor macros with C++11 features like generalized constant expressions, perfect forwarding, and lambda expressions. This transition enhances code clarity and maintainability, supporting modern software design. However, adoption faces challenges, particularly in automating refactoring without breaking functionality, as some macros reference free variables or recursive instantiations. The study emphasizes the need for careful analysis and iterative refinement to ensure a successful transformation (Kumar et al., 2012).

Dantas et al. investigated Java legacy system evolution by applying 2,462 transformations to 40 open-source projects via pull requests. Their findings show that transformations improving readability are generally accepted, while substantial changes may be rejected. This underscores the importance of code clarity and highlights the subjective nature of perceived improvements, especially when mixed idioms are involved and functional programming styles are less familiar to some developers (Dantas et al., 2018).

Gokhale et al. introduced a static analysis technique for refactoring JavaScript applications from synchronous to asynchronous APIs using async/await, aiming to minimize code disruption. Evaluating 12 applications with 316 API calls, their tool, Desynchronizer, successfully refactored 244 out of 256 candidates. While asynchronous APIs enhance performance, responsiveness, and scalability, challenges include higher syntactic complexity and potential behavioral changes, emphasizing the need for thorough testing during migration (Gokhale et al., 2021).

Finally, Nicolini et al. examined JavaScript feature adoption via transpilers, analyzing 1,000 open-source projects to assess Babel plug-in usage and adoption challenges. They find that 73 projects use at least one new JavaScript proposal, relying on Babel to handle compatibility. With 86% average browser support, transpilers play a key role in modern web development. Developers adopt new features for performance and early adoption, but face cross-browser compatibility issues and integration complexity (Nicolini et al., 2024).

7.2 Feature Adoption and Usage Analysis

After the release of new programming language versions that introduce modern features, researchers began analyzing how developers adopt these modern features. Studies on feature adoption in languages like Kotlin, TypeScript, Python, C++, Java and JavaScript explore when and how developers integrate these features, as well as the challenges that hinder their adoption. For example, Mateus and Martinez investigated Kotlin feature adoption in An-

droid development using 387 open-source applications, finding that 15 out of 26 features are frequently used, with type inference, lambda, and safe call being the most prevalent. Essential features are often adopted early, while less common ones appear later. However, developers face challenges due to the lack of benchmarks and training materials, limiting the adoption of features like property delegation. The study highlights the need for better tutorials to encourage broader use of advanced Kotlin capabilities (Mateus and Martinez, 2020).

Peng et al. analyzed Python feature usage across 35 projects from eight domains, highlighting developer preferences for simple and safe features like safety checks, testing, and dynamic capabilities while avoiding complex constructs such as heterogeneous lists and diamond inheritance. Exception handling, decorators, and nested classes/functions are frequently used, with a focus on custom decorators and ImportError handling. However, modern features like gradual typing and keyword-only arguments face low adoption due to complexity and potential errors, despite their benefits for safety and performance (Peng et al., 2021).

Scarsbrook et al. analyzed 454 open-source TypeScript repositories to examine feature adoption trends over three years. They find that type modifiers on import and template literal types are frequently adopted, with over 20 repositories using them within a year, while features like static blocks in classes see much lower adoption. Despite frequent compiler updates, many language features are adopted more slowly, suggesting that developers prioritize compiler upgrades over new features, possibly due to complexity or perceived utility (Scarsbrook et al., 2023).

Uesbeck et al. analyzed the impact of C++11 lambda expressions through a randomized controlled trial, comparing them with iterators among students and professional developers. The study finds no significant benefits in time efficiency or error reduction, with students facing more compiler errors and longer task completion times. Developers reported debugging difficulties, complicating adoption in areas like concurrent programming. While lambdas can simplify code, they may introduce unintended complexities, suggesting the need for further research on their practical impact (Uesbeck et al., 2016).

Chen et al. investigated C++ template usage across 1,267 revisions from 50 open-source systems, finding that templates mainly prevent code duplication, but their benefits are limited to a few frequently used cases. Function templates do not fully replace C-style generics, and developers with a C background show no strong preference between them. Adoption of modern C++11 templates varies, with variadic class templates frequently used, while variadic function templates and alias templates see lower adoption. The study highlights compatibility challenges and the need for better tooling to support modern template adoption (Chen et al., 2020).

Lucas et al. examined C++ rejuvenation practices in KDE projects, finding growth adoption of lambda expressions, range-based for loops, and auto-typed variables in over 60% of projects, while modern multi-threading features remain rarely used. Developers leverage tools like Clazy and Clang-Tidy to

facilitate large-scale updates, driven by goals of improving readability, conciseness, and attracting new contributors. However, challenges such as high update costs, framework incompatibilities, and resistance to change hinder the adoption of modern features (Lucas et al., 2023).

Dyer et al. analyzed 31k open-source Java projects, revealing varied adoption rates of new language features, with enhanced-for loops and annotations being the most popular. The study finds that some features are adopted before their official release, showing developer anticipation and readiness. Motivations for adoption include improving code efficiency and clarity, while challenges such as lack of awareness and training hinder broader usage (Dyer et al., 2014a).

Mazinanian et al. investigated lambda expression adoption in Java using 241 open-source projects and surveys from 97 developers. The study finds a two-fold increase in lambda usage from 2015 to 2016, driven by motivations like improving readability, reducing duplication, and enabling lazy evaluation. However, challenges include inefficient use of functional interfaces and difficulties adapting to a functional programming mindset (Mazinanian et al., 2017).

Similarly, Lucas et al. examined the impact of lambda expressions on Java program comprehension through a mixed-method study, analyzing 66 code snippet pairs. The findings reveal a contradiction: while quantitative analysis found no readability improvement, qualitative insights suggest enhanced comprehension. Developers adopt lambdas for simplified code, but challenges include understanding functional programming and the cognitive load of transitioning from traditional constructs (Lucas et al., 2019).

Zheng et al. analyzed the removal of lambda expressions in Java, analyzing 117 issues from Apache JIRA, code commits, GitHub issues, and a user study. The findings show that performance issues and memory leaks are key reasons for removal. Developers remove lambdas to simplify code and prevent bugs, but face challenges in managing inappropriate lambda usage that introduces side effects (Zheng et al., 2021).

Silva et al. examined class usage in JavaScript, finding that 40% of systems rely on class-based design, while prototype-based inheritance remains unpopular. Using static analysis on 50 GitHub projects, they show that developer experience influences class adoption more than system size. Challenges include prototype complexity and the lack of encapsulation mechanisms, highlighting the need for better tools to support class-based development (Silva et al., 2015).

Alves et al. analyzed functional programming adoption in JavaScript, examining 91 open-source repositories and trends in recursion, immutability, lazy evaluation, and functions as values. The study finds a 255% increase in immutability-related structures and a rise in lazy evaluation and higher-order functions, while recursion and callbacks declined. Functional structures are less likely to be removed in bug-fixing commits, suggesting lower error-proneness. Developers adopt these features for modularity, clarity, and maintainability, but refactoring existing systems remains a key challenge (Alves et al., 2022).

Our study reveals the early and frequent adoption of modern JavaScript features, such as const declarations and arrow functions, reinforcing prior research that highlights similar adoption patterns in JavaScript (Silva et al., 2015) and other programming languages like Kotlin (Mateus and Martinez, 2020), TypeScript (Scarsbrook et al., 2023), and Java (Mazinanian et al., 2017). Motivations for adopting these features include improved code readability, conciseness, and maintainability. However, the variability in feature usage—such as the high adoption of Template String Expressions and the lower adoption of Await Operators—highlights contextual differences influenced by tooling, compatibility issues, and developer familiarity (Gokhale et al., 2021; Scarsbrook et al., 2023).

Challenges in adopting modern features are consistent with findings across programming languages, including compatibility issues with older constructs (Lucas et al., 2023), performance concerns (Kumar et al., 2012), and the cognitive burden of adopting functional programming paradigms (Alves et al., 2022). Transpilers like Babel play a pivotal role in facilitating feature adoption by addressing cross-browser compatibility but introduce trade-offs, such as increased code complexity and potential bugs (Nicolini et al., 2024). These findings emphasize the need for tools and strategies that balance modern feature adoption with the practical constraints of legacy systems.

This study advances the literature on JavaScript by providing a comprehensive analysis of the adoption patterns, motivations, and challenges associated with modern language features introduced from ES6 onwards. It highlights trends of early adoption, the critical role of transpilers like Babel, and the nuanced decision-making developers face in balancing modern feature integration with legacy constraints. By identifying gaps in adoption, such as low uptake of Await Operators and challenges in functional programming, the study offers insights that complement findings in other languages, emphasizing the unique dynamics of JavaScript's ecosystem.

8 Conclusions and Future Works

This paper has explored the general question related to the adoption of modern JavaScript features—that is, features introduced in the sixth version of the ECMAScript standard (ES6) onwards. The research brings insights into the adoption patterns, timeline, and trends of modern JavaScript features.

The reliance on modern JavaScript features in open-source repositories is quite high for certain modern features, with a varied adoption pattern for others. Developers tend to adopt new features early, reflecting a forward-thinking approach to utilizing the language's evolving capabilities. The adoption trends of modern JavaScript features demonstrate a clear pattern of gradual uptake following their official introduction in ECMAScript versions. While some features are quickly integrated into development practices, others require a maturation period before seeing significant usage. This behavior aligns with broader

trends observed in the adoption of new programming language features across different development communities (Dyer et al., 2014b; Lucas et al., 2023).

The exploration of motivations and challenges in rejuvenating legacy JavaScript code reveals a complex landscape where developers balance the benefits of adopting modern language features with practical impediments. Improving code comprehension emerges as a primary motivation, driven by the desire to enhance conciseness and maintainability through features like Arrow Function Declarations, Async Declarations, and Await Operators. However, pervasive compatibility issues with older browsers, libraries, and runtime environments present formidable challenges. Resistance to change rooted in existing code conventions, concerns over performance impacts, and the perceived complexity of new syntax further complicate adoption efforts. These findings underscore the nuanced decision-making process developers face when rejuvenating codebases, emphasizing the need for pragmatic strategies that reconcile technological advancement with practical constraints.

Regarding the findings related to the transpiler-based approach for rejuvenating JavaScript code, it is evident that while tools like Babel facilitate the adoption of modern language features, they introduce complexities and potential challenges. Developers utilize transpilers to bridge compatibility gaps between modern JavaScript syntax and legacy environments, enabling immediate integration of advanced features like Arrow Function Declarations, Async Declarations, and Await Operators constructs. However, our analysis highlights significant trade-offs, as transpilation can lead to increased code complexity and potential bugs, illustrated by issues in Babel's handling of Arrow Function Declarations and performance overhead concerns.

Additionally, our analysis of rejuvenation efforts in JavaScript code revealed growth adoption of modern language features across various projects, reflecting a concerted effort to enhance code quality and maintainability. These rejuvenation efforts collectively contribute to modernize JavaScript codebases, enhancing overall software quality, yet underscore the need for careful consideration and strategic implementation of transpiler-based approaches to ensure effective code rejuvenation efforts in JavaScript projects.

For future work, we identified two opportunities to delve deeper into the factors influencing the adoption patterns of programming language features. By looking at how things like community support, the availability of tools, and the quality of the documentation affect the use of a feature, researchers can find out what makes people adopt it.

Additionally, another interesting direction for future work would be to investigate whether the limited adoption of certain modern JavaScript features is influenced by a lack of external support, such as compatibility with libraries, runtime environments, or less common browsers. While our study acknowledges that major browsers fully support the features analyzed, the extent to which external constraints impact feature adoption remains unclear. A dedicated study exploring these potential barriers could provide deeper insights into the challenges open-source developers face and further contextualize the adoption trends observed in our analysis.

These opportunities for future research may contribute to our understanding of software evolution and programming language development dynamics.

Acknowledgements We also thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and helpful suggestions.

Author Contributions *Walter Lucas, Rafael Nunes, Rodrigo Bonifácio, Fausto Carvalho, and Michael Silva* - Conceptualization, Data curation, Methodology, Software, Formal analysis, Investigation, Writing - Original Draft, Visualization, Supervision and *Ricardo Lima, Adriano Torres, Paola Accioly, Eduardo Monteiro, and João Saraiva* Data curation, Supervision, Writing - review & editing.

Funding Not applicable.

Ethical approval and Informed consent Not applicable.

Code and Data Availability Our study is fully reproducible. All developed tools, collected data, and Python scripts are available online for statistical analysis in (Mendonça, 2025).

Declarations

Conflicts of interests The authors declare that they have no conflict of interest.

References

Alves F, Oliveira D, Madeiral F, Castor F (2022) On the bug-proneness of structures inspired by functional programming in javascript projects. DOI 10.48550/arXiv.2206.08849

Andreasen E, Gong L, Møller A, Pradel M, Selakovic M, Sen K, Staicu C (2017) A survey of dynamic analysis and test generation for javascript. ACM Comput Surv 50(5):66:1–66:36, DOI 10.1145/3106739, URL https://doi.org/10.1145/3106739

Avelino G, Constantinou E, Valente MT, Serebrenik A (2019) On the abandonment and survival of open source projects: An empirical investigation. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 1–12, DOI 10.1109/ESEM.2019.8870181

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Notkin D, Cheng BHC, Pohl K (eds) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, IEEE Computer Society, pp 712–721, DOI 10.1109/ICSE.2013.6606617, URL https://doi.org/10.1109/ICSE.2013.6606617

Bogachenkova V, Nguyen L, Ebert F, Serebrenik A, Castor F (2022) Evaluating atoms of confusion in the context of code reviews. In: IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022, IEEE, pp 404–408, DOI 10.1109/ICSME55016.2022.00048, URL https://doi.org/10.1109/ICSME55016.2022.00048

Brito A, Hora AC, Valente MT (2022) Towards a catalog of composite refactorings. CoRR abs/2201.04599, URL https://arxiv.org/abs/2201.04599, 2201.04599

Cantelon M, Harter M, Holowaychuk T, Rajlich N (2013) Node.Js in Action, 1st edn. Manning Publications Co., USA

Chen L, Wu D, Ma W, Zhou Y, Xu B, Leung H (2020) How c++ templates are used for generic programming: An empirical study on 50 open source systems. ACM Trans Softw Eng Methodol 29(1), DOI 10.1145/3356579, URL https://doi.org/10.1145/3356579

Clarke V, Braun V (2017) Thematic analysis. The journal of positive psychology 12(3):297–298

Cleveland WS (1979) Robust locally weighted regression and smoothing scatterplots. Journal of the American statistical association 74(368):829–836

Dabic O, Aghajani E, Bavota G (2021) Sampling projects in github for MSR studies. In: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, IEEE, pp 560–564

Daka E, Campos J, Fraser G, Dorn J, Weimer W (2015) Modeling readability to improve unit tests. In: Nitto ED, Harman M, Heymans P (eds) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, ACM, pp 107–118, DOI 10.1145/2786805.2786838, URL https://doi.org/10.1145/2786805.2786838

Danial A (2021) cloc: v1.92. DOI 10.5281/zenodo.5760077, URL https://doi.org/10.5281/zenodo.5760077

Dantas R, Carvalho A, Marcilio D, Fantin L, Silva U, Lucas W, Bonifácio R (2018) Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In: Oliveto R, Penta MD, Shepherd DC (eds) 25th

International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, IEEE Computer Society, pp 497–501, DOI 10.1109/SANER.2018.8330247, URL https://doi.org/10.1109/SANER.2018.8330247

Dyer R, Rajan H, Nguyen HA, Nguyen TN (2014a) Mining billions of ast nodes to study actual and potential usage of java language features. In: Proceedings of the 36th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE 2014, p 779–790, DOI 10.1145/2568225.2568295, URL https://doi.org/10.1145/2568225.2568295

Dyer R, Rajan H, Nguyen HA, Nguyen TN (2014b) Mining billions of AST nodes to study actual and potential usage of java language features. In: Jalote P, Briand LC, van der Hoek A (eds) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, ACM, pp 779–790, DOI 10.1145/2568225.2568295, URL https://doi.org/10.1145/2568225.2568295

Ecma E (1997) 262: Ecmascript language specification. Tech. rep., ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,, URL https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

Flanagan D (2020) JavaScript: The Definitive Guide : Master the World's Most-used Programming Language. O'Reilly Media, Incorporated, URL https://books.google.com.br/books?id=b5G4zAEACAAJ

Gokhale S, Turcotte A, Tip F (2021) Automatic migration from synchronous to asynchronous javascript apis. Proc ACM Program Lang 5(OOPSLA), DOI 10.1145/3485537, URL https://doi.org/10.1145/3485537

Haverbeke M (2014) Eloquent JavaScript: A Modern Introduction to Programming, 2nd edn. No Starch Press, USA

Huang C, Petukhina A (2022) Applied Time Series Analysis and Forecasting with Python. Statistics and Computing, Springer International Publishing, URL https://books.google.com.br/books?id=TfaVEAAAQBAJ

Keith J (2005) DOM Scripting. Apress, DOI 10.1007/978-1-4302-0062-8

Kelhini FK (2021) Modern Asynchronous JavaScript. Pragmatic Bookshelf, URL https://www.amazon.com/dp/B09NQQ8715

Kumar A, Sutton A, Stroustrup B (2012) Rejuvenating c++ programs through demacrofication. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp 98–107, DOI 10.1109/ICSM.2012.6405259

Lin HY, Thongtanunam P (2023) Towards automated code reviews: Does learning code structure help? In: Zhang T, Xia X, Novielli N (eds) IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023, IEEE, pp 703–707, DOI 10.1109/SANER56733.2023.00075, URL https://doi.org/10.1109/SANER56733.2023.00075

Lucas W, Bonifácio R, Canedo ED, Marcilio D, Lima F (2019) Does the introduction of lambda expressions improve the comprehension of java programs? In: do Carmo Machado I, Souza R, Maciel RSP, Sant'Anna C (eds) Proceed-

ings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019, ACM, pp 187–196, DOI 10.1145/3350768.3350791, URL https://doi.org/10.1145/3350768.3350791

Lucas W, Carvalho F, Nunes RC, Bonifácio R, Saraiva J, Accioly P (2023) Embracing modern c++ features: An empirical assessment on the kde community. Journal of Software: Evolution and Process n/a(n/a):e2605, DOI https://doi.org/10.1002/smr.2605, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2605, https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2605

Mateus BG, Martinez M (2020) On the adoption, usage and evolution of kotlin features in android development. In: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Association for Computing Machinery, New York, NY, USA, ESEM '20, DOI 10.1145/3382494.3410676, URL https://doi.org/10.1145/3382494.3410676

Mazinanian D, Ketkar A, Tsantalis N, Dig D (2017) Understanding the use of lambda expressions in java. Proc ACM Program Lang 1(OOPSLA), DOI 10.1145/3133909, URL https://doi.org/10.1145/3133909

Mendonça WL (2025) Understanding the adoption of modern javascript features: An empirical study on open-source systems. DOI 10.5281/zenodo.14796287, URL https://doi.org/10.5281/zenodo.14796287

Morgan J, Stewart A (2018) Simplifying JavaScript: Writing Modern JavaScript with ES5, ES6, and Beyond. The Pragmatic Programmers, Pragmatic Bookshelf, URL https://books.google.com.br/books?id=SIyAtAEACAAJ

Nicolini T, Hora AC, Figueiredo E (2024) On the usage of new javascript features through transpilers: The babel case. IEEE Softw 41(1):105–112, DOI 10.1109/MS.2023.3243858, URL https://doi.org/10.1109/MS.2023.3243858

Oliveira D (2021) Recommending code understandability improvements based on code reviews. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021 - Workshops, Melbourne, Australia, November 15-19, 2021, IEEE, pp 131–132, DOI 10.1109/ASEW52652.2021.00035, URL https://doi.org/10.1109/ASEW52652.2021.00035

Oliveira D, Bruno R, Madeiral F, Castor F (2020) Evaluating code readability and legibility: An examination of human-centric studies. In: IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020, IEEE, pp 348–359, DOI 10.1109/ICSME46990.2020.00041, URL https://doi.org/10.1109/ICSME46990.2020.00041

Parr T (2013) The definitive antlr 4 reference. The Definitive ANTLR 4 Reference pp 1–326

Peng Y, Zhang Y, Hu M (2021) An empirical study for common language features used in python projects. In: 28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021, IEEE, pp 24–35, DOI 10.1109/SANER50967.2021.00012, URL https://doi.org/10.1109/SANER50967.2021.00012

Phang CL (2022) ECMAScript 2022: The Definitive Guide to Modern JavaScript. Self-published, URL https://www.amazon.com/ECMAScript-2022-Definitive-Modern-JavaScript/dp/B0BLG6SWRT

Pirkelbauer P, Dechev D, Stroustrup B (2010) Source code rejuvenation is not refactoring. In: van Leeuwen J, Muscholl A, Peleg D, Pokorný J, Rumpe B (eds) SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings, Springer, Lecture Notes in Computer Science, vol 5901, pp 639–650, DOI 10.1007/978-3-642-11266-9_53, URL https://doi.org/10.1007/978-3-642-11266-9_53

Rauschmayer A (2015) Exploring ES6: Upgrade to the Next Version of JavaScript. Leanpub, URL https://exploringjs.com/es6/

Rauschmayer A (2024) Exploring JavaScript. Self-published, URL https://exploringjs.com/js/index.html

Resig J, Bibeault B, Maras J (2016) Secrets of the JavaScript Ninja, Second Edition. ITpro collection, Manning Publications, URL https://books.google.com.br/books?id=dq16zQEACAAJ

Rosenkranz S, Staegemann D, Volk M, Turowski K (2024) Explaining the business-technological age of legacy information systems. IEEE Access 12:84579–84611, DOI 10.1109/ACCESS.2024.3414377, URL https://doi.org/10.1109/ACCESS.2024.3414377

Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: a case study at google. In: Paulisch F, Bosch J (eds) Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, pp 181–190, DOI 10.1145/3183519.3183525, URL https://doi.org/10.1145/3183519.3183525

Saldana J (2012) The Coding Manual for Qualitative Researchers. English short title catalogue Eighteenth Century collection, SAGE Publications, URL https://books.google.com.br/books?id=kUms8QrE_SAC

Scarsbrook JD, Utting M, Ko RKL (2023) Typescript's evolution: An analysis of feature adoption over time. 2303.09802

Silva D, da Silva JP, de Souza Santos GJ, Terra R, Valente MT (2021) Refdiff 2.0: A multi-language refactoring detection tool. IEEE Trans Software Eng 47(12):2786–2802, DOI 10.1109/TSE.2020.2968072, URL https://doi.org/10.1109/TSE.2020.2968072

Silva LH, Ramos M, Valente MT, Bergel A, Anquetil N (2015) Does javascript software embrace classes? In: Guéhéneuc Y, Adams B, Serebrenik A (eds) 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, IEEE Computer Society, pp 73–82, DOI 10.1109/SANER.2015.7081817, URL https://doi.org/10.1109/SANER.2015.7081817

Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history. In: Chaudron M, Crnkovic I, Chechik M, Harman M (eds) Proceedings of the 40th In-

ternational Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, pp 483–494, DOI 10.1145/3180155.3180206, URL https://doi.org/10.1145/3180155.3180206

Uesbeck PM, Stefik A, Hanenberg S, Pedersen J, Daleiden P (2016) An empirical study on the impact of C++ lambdas and programmer experience. In: Dillon LK, Visser W, Williams LA (eds) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, pp 760–771, DOI 10.1145/2884781.2884849, URL https://doi.org/10.1145/2884781.2884849

Vishwas B, PATEL A (2020) Hands-on Time Series Analysis with Python: From Basics to Bleeding Edge Techniques. Apress, URL https://books.google.com.br/books?id=cTeHzQEACAAJ

Wirfs-Brock A, Eich B (2020) Javascript: The first 20 years. Proc ACM Program Lang 4(HOPL), DOI 10.1145/3386327, URL https://doi.org/10.1145/3386327

Zhao Y, Serebrenik A, Zhou Y, Filkov V, Vasilescu B (2017) The impact of continuous integration on other software development practices: a large-scale empirical study. In: Rosu G, Penta MD, Nguyen TN (eds) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE Computer Society, pp 60–71, DOI 10.1109/ASE.2017.8115619, URL https://doi.org/10.1109/ASE.2017.8115619

Zheng M, Yang J, Wen M, Zhu H, Liu Y, Jin H (2021) Why do developers remove lambda expressions in java? In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, IEEE, pp 67–78, DOI 10.1109/ASE51524.2021.9678600, URL https://doi.org/10.1109/ASE51524.2021.9678600

Walter Lucas is a PhD candidate in Computer Science at the University of Brasília since 2019, holding an MSc from UNB and a Bachelor's in Information Systems. From 2020 to 2022, he contributed to a collaborative R&D project with FAPDF/UNB/CNJ. His work focuses on software engineering, code comprehension, software evolution, and program transformations, with over five years of R&D experience.

Rafael Nunes is a Computer Science BSc undergraduate at the University of Brasília. Currently working as a Software Engineer for the Payment Industry, he's interested in studying computer networks, compilers, and distributed systems.

Rodrigo Bonifácio is an associate professor at the Computer Science Department, University of Brasília, Brazil, and a member of the Software Productivity Group, led by Prof. Paulo Borba. His research interests lie in understanding the common usage of programming language features, program transformations, software architecture and modularity, and software security.

Fausto Carvalho is a Master's student in Informatics at the University of Brasília with a B.Sc. in Computer Science and more than 24 years of experience in software engineering across Brazil and Canada. Research interests include program analysis, software security, fuzzing, and programming languages.

Ricardo Lima is a Ph.D. candidate in Computer Science at Universidade de Brasília. His research interests include artificial intelligence, software engineering and digital government. He works as General Coordinator of Digital Economy, at the Ministry of Industry in Brazil, working with the formulation and coordination of public policies related to the Digital Economy, with a focus on the digital transformation of the Brazilian industrial sector.

Mchael Silva is a PhD student in Machine Learning applied to Software Engineering at the University of Brasilia. He holds a Master's degree in Computer Science from the same university where he studied the application of machine learning to improve user experience in banking systems. He is currently a Senior Application Architect at Amazon focusing on the financial services area.

Adriano Torres is a Master of Philosophy graduate at the University of Adelaide and a PhD candidate at the University of Brasilia. Whilst pursuing his academic interests, Adriano has kept working in both startups and big corporations in the software industry throughout his entire career. He is a strong believer that in the software industry, theory should follow mostly from practice, and that it is essential for software engineering research to both seek to involve actual practitioners and to learn from them. His main interests are programming languages and compilers.

Paola Accioly is a professor at the Center for Informatics of the Federal University of Pernambuco, Brazil. Her research interests include support for collaborative software development and empirical software engineering. She holds a PhD in Computer Science from the Federal University of Pernambuco and has previously worked at the Federal University of Cariri.

Eduardo Monteiro is a professor in the Department of Statistics at the University of Brasília. His research focuses on applied statistics, particularly in the domains of technology, agriculture, and public sector data analysis.

João Saraiva is an Associate Professor at the department of Informatics, University of Minho, Portugal, and a senior research member of HASLab/INESC TEC. He obtained a Ph.D. degree in Computer Science from Utrecht University in 1999. His main research contributions have been in the field of programming language design and implementation, green software, and functional programming.

A Appendix

Table 5 Feature adoption in various projects by source code rejuvenation with corresponding commit details and quantities.

Project name	commit link	Feature adopted & Qtd
bookreader	a2f4c92e7	Arrow Function = 4
bookreader	17beab4b4	Const = 5
bookreader	d21cc72d5	Arrow Function = 1
chrome-extensions-samples	0672807d4b	Optional Chaining = 1
cinnamon	fb0c23a363	Default Parameter = 1, Spread Statements = 1
client-nodejs	ad024e1a4a0	Arrow Function = 1
converse.js	654e72baad8	Optional Chaining = 2
ember.js	f8812ee1d7	Class = 4
encoded	0d9dd05be0	Optional Chaining = 1
encoded	3ce1713736	Import = 1, Export = 2
hexo	1ff82654cf	Arrow Function = 1
hexo	4d071c4102	Arrow Function = 3
highlight.js	a69eb6a2e	Arrow Function = 92
jasmine	1166d10e4	Const = 2038, Let = 152
modernizr	4c263c258	Import = 5, Export = 3
node-postgres	039b5baef6e	Arrow Function = 1
novnc	cdb860ad84	Const = 5, Let = 7
novnc	2b5f94fa6a	Const = 681, Let = 214
novnc	0e4808bf6f3	Class = 10
novnc	651c23ece37	Arrow Function = 111
openpgpjs	5142003b09	Await Declarations = 6, Object Destructuring = 6
pouchdb	f30b9fbab9e2	Const = 1, Let = 1, Arrow Function = 2, Async = 1, Await = 2
pouchdb	99c24a5861e	Import = 341
qunit	f0da7b88197	Arrow Function = 1, Default Parameter = 1
qunit	df70c00a864	Spread Statements = 1
qunit	dc2887b7fe7	Arrow Function = 2
sefaria-project	182f74872e	Default Parameter = 3, Object Destructuring = 1
travis-web	0e6dc8db4e	Arrow Function = 1
travis-web	4cac82832f	Arrow Function = 1
video.js	9e76477b492	Arrow Function = 1
video.js	ffac358bf745a	Arrow Function = 1