ANALYZING THE IMPACT ON IDENTIFIABLE DEFECTS IN A CODE INDEPENDENT OF TYPESCRIPT

G.M. Deshan Madurajith
(199343H)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

June 2022

ANALYZING THE IMPACT ON IDENTIFIABLE DEFECTS IN A CODE INDEPENDENT OF TYPESCRIPT

G.M. Deshan Madurajith
(199343H)

Thesis/Dissertation submitted in partial fulfilment of the requirements for the degree Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

June 2022

DECLARATION OF THE CANDIDATE & SUPERVISOR

I declare that this is my own work and this thesis does not incorporate without acknowledgment of any material previously submitted for a Degree or Diploma in any other universities or institutes of higher learning and to the best of my knowledge and belief does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic, or another medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Name of Candidate: G.M. Deshan Madurajith

Signature: ..... Date: ……

The above candidate has carried out research for the Master' thesis under my supervision.

Name of the supervisor: Prof. Indika Perera

Signature: ..... Date: …….

Abstract

Javascript has become the most widespread programming language used to build software for many platforms, including web, mobile, backends, hardware, and desktop applications. It is a dynamically typed programming language that gives freedom to ignore types and build applications quickly. Still, Javascript is a poor language for identifying bugs at the compile time and maintaining large applications because of the dynamic behavior. The developers can use Typescript to add type syntaxes on top of JavaScript as a solution.

However, not enough empirical studies exhibit how Typescript impacts detecting detects in applications. We decided to follow an empirically quantified "what-if" style of experimentation. We collected merged javascript bug-fix pull requests from selected open-source projects. Then we selected candidate bugs by applying our predefined criteria such as discarded pull requests with more than five files, code-refactoring, and configuration changes. We manually added Typescript annotation to the buggy code base and checked whether Typescript could detect the defects at the compile type. We assessed 500 bug-fix pull requests over five projects and identified that using Typescript over Javascript could have prevented 22.7% of bugs.

According to our literature review, this is one of the few studies related to Typescript identifying bug impact. We believe this study will be significant evidence to consider using Typescript over Javascript in the future to reduce the significant number of bugs. This result will influence developers to adapt to the Typescript from Javascript. However, Typescript is not a silver build for Javascript because developers have to add extra code and complexity to the codebase. So, More research on Typescript is needed. In the future, we plan to explore the cognitive complexity of applying Typescript and the number of required lines to annotate. Furthermore, we plan to use a Typescript converter for annotating to increase the number of sample bugs to analyze.

Keywords: Typescript, Javascript, static typed, dynamic typed

ACKNOWLEDGEMENT

First, I am indebted to my thesis advisor, Prof. Indika Perera. He helped to finalize the scope and advised me on how the contents and research should be. I also thank all the lecturers who teach me because those subjects helped me a lot to choose this research. Finally, yet importantly, I want to mention the role of my parents in my journey. They add unconditional support to manage my career & studies in a free mindset.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

According to the stats, Javascript (JS) is one of the most popular and widely programming languages that supports building software for the web, mobile, backend, hardware, and desktop applications. It is a dynamic programming language that gives freedom to ignore types and build applications quickly.

Usually, it is known as dynamically typed languages are great for implementing small-scale applications [1]. Although Javascript is a beginner-friendly language, it often leads to poor software quality [1, 2, 3, 4] Equal to beliefs about other scripting languages like PHP [5]. However, it becomes challenging when the application becomes complex and extensive. It increases the possibility of making mistakes, and fixing them can make another issue. Sometimes it takes a long time to track or understand the code implementation.

The developers cannot effortlessly see bugs without running the application when identifying basic errors such as incompatibilities and type errors because of dynamic behavior. As a solution, Microsoft introduced TypeScript, and Facebook introduced Flow which provides a static typing system for javascript that helps developers get warnings or errors at the compile time. In other words, the IDE with correct plugins shows errors while typing the codes. As a cost, the developers need to spend extra effort to write Types, and some need more time to understand how to write types correctly.

However, there are not enough studies or evidence to provide empirical evidence to support that TypeScript leads to the detection of more bugs in the early stage of development. Many studies claim that statically typed language is better for software quality than dynamically typed ones [6, 7]. Some studies say that developers can easily see errors [8], which helps to make better documentation because of the type of system [9].

Since there is not enough research to prove whether typed Javascript can prevent bugs earlier than Javascript, we decided to examine this area. Although there are a few options to provide type systems for the Javascript application, we decided to select only Typescript because it is the most popular static typing.

1.1 Motivation

Today, the development process is not a simple task because the developers have many things to do before and after writing a code. Once they build a feature, they need to maintain the code base for a longer time. When projects become larger day by day, the Javascript projects naturally make numerous bugs like validation, null checks, and undefined issues because of the dynamic behavior. The bug causes the end-user unhappy, and it adds a huge cost to the companies.

According to the survey [10] done by stepsize.com, they claim that 62% of people quit their job because of the code quality and the codebase's health. Since hiring and retaining developers are expensive these days, the company needs to maintain a healthy codebase.

According to the research and experiences from many companies, Typescript can provide huge benefits to the Javascript developers to avoid many bugs and improve the developer experience.

Airbnb has migrated its entire Javascript codebase into Typescript to minimize the bugs and improve the developer experience in 2019 [2]. According to their postmortem analysis of bugs, they have identified that they could reduce the bugs in the code by 38% [19]. Rodoslav Kirov and Bowen Ni, who work as developers at google, say that "using Typescript is simple and pleasant for all google engineers" [11]. One famous chat application company called Slack also says that converting to Typescript could find several small bugs [12].

Apart from the experiences from the big tech companies related to converting to Typescript, the Stack Overflow trends [13] show the exponential growth in Typescript in the past five years while Javascript is not growing but staying at the same level.Figure 1.1

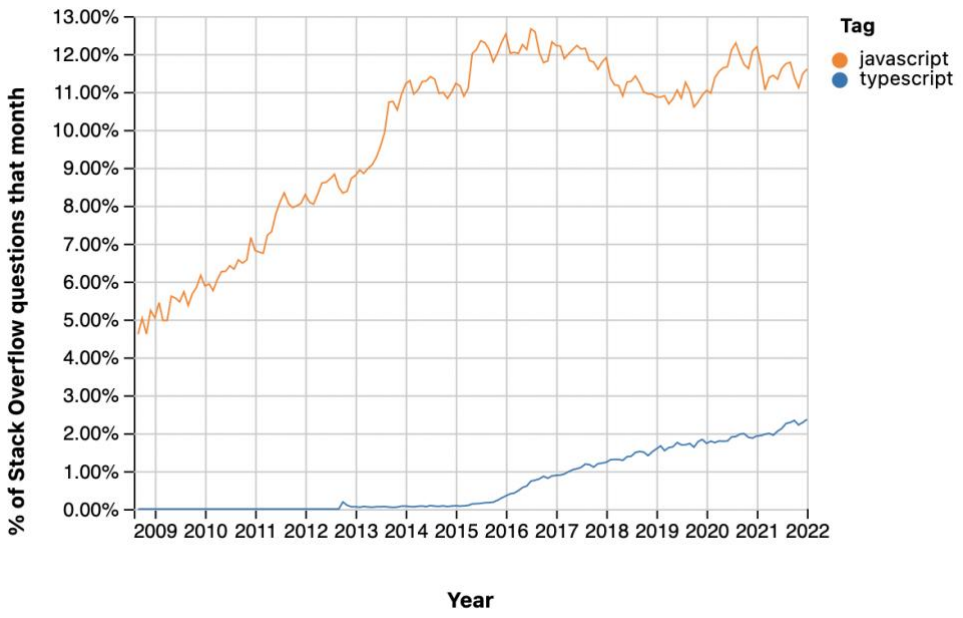


Figure 1.1 Javascript vs Typescript trends. Adapted from [13]

By default, most Javascript support Typescript as the default configuration, such as Angular, Ionic, and NestJs. Other popular frameworks and libraries such as React, React Native, VueJs support Typescript configuration. Why does that framework support migration or use Typescript by default? Does it help to increase productivity or reduce the bug rate?

Due to those reasons, we were curious to research about how it impacts the bug detection. Then we can use this report as a reference for companies to either convert or start new projects with Typescript.

1.2 Aim and objective

There is no proper evidence why the developers choose Typescript or Javascript for their development. So, we assume they choose it depending on their knowledge, experience, intuition, or other reason.

Today, most of the time, developers follow loosely coupled solutions such as Mircsorvices, Serverless, Mico-Front end, and Modularizations. Also, some developers write separate utility libraries apart from their main projects. So, the developers get an opportunity to start a new project, create a new component for loosely coupled architecture or refactor the existing code. When selecting a language

(Typescript or Javascript) for those projects, they consider different factors such as productivity, maintainability, efficiency, portability, and tool support [14]. Also, it is essential to reduce the bugs as much as possible early because more defects mean low quality and are expensive to maintain and continue the project.

This research answers, "How impactful is it to use Typescript over Javascript to detect defects?". We believe this research result will be helpful for the developers and stakeholders to choose between Typescript and Javascript.

1.3 Scope

The scope of this research is limited to Javascript and Typescript. Even though there is another competitive type framework called "Flow" from Facebook, their usage is quite low. The Figure 1.2 depicts that Flow usage is very low compared to Typescript [15].

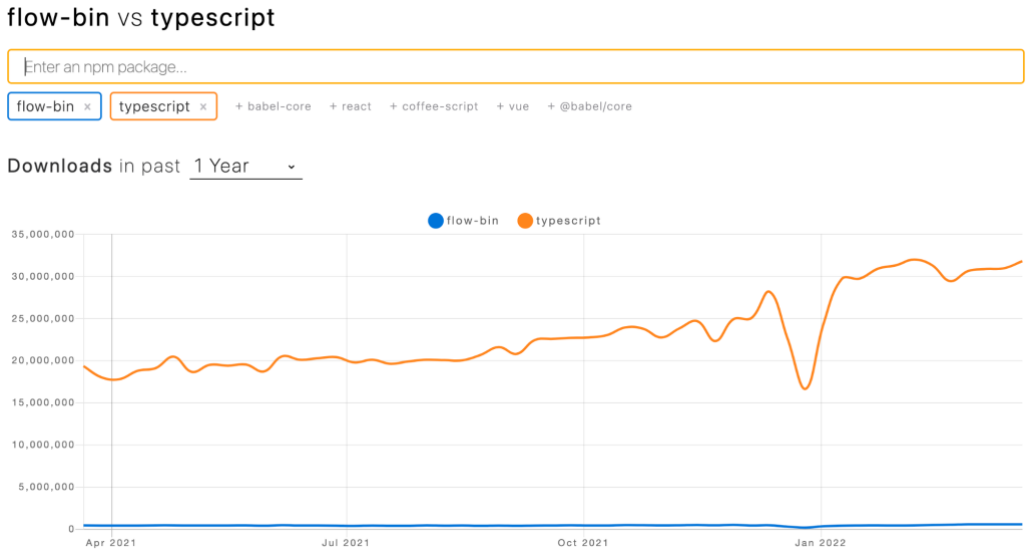


Figure 1.2 Flow-bin vs Typescript trends. Adapted from [15]

This research selected only five public active projects, and we selected them by considering factors such as the number of Github stars, start dates, and the number of pull requests. Since our main objective is to answer "How impactful is it to use Typescript over Javascript to detect defects?". We considered only the merged bugs and how Typescript can identify those bugs at the compile time. In this research, we did not consider cognitive complexity's impact and time spent using Typescript over

Javascript. We also did not consider the engineers' experience and skillfulness and How Typescript best practices impact the bug.

1.4 Structure of the thesis

Chapter 1 briefly introduces the project background, objective, and scope. Chapter 2 discusses the background for the literature review under four sections: background, bug-related studies, Bug collections & classifications techniques, and Quantifying and analyzing detectable bugs. Under the bug-related studies topic, we discuss five studies closely related to Typescript or bug-related research. Chapter 3 provides the method for selecting public projects, bug collection, bug classifications, converting to Typescript, and quantifying the identifiable defects compared to Typescript. Chapter 4 discusses the results, challenges, and efforts. Chapter 5 discusses the summary and conclusion.

2. LITERATURE REVIEW

2.1 Background

Based on the recent surveys from Stack Overflow [13] and Node.js foundation [16], Javascript has become the most popular language, but still, it is error-prone due to its dynamic, asynchronous nature, and loosely-typed nature.

Today, most bug identification solutions do not supply sufficient warnings about the faults in the code [17]. Most of those tools and research studies are designed and developed based on empirical methods (observations or experience rather than theory/pure logic) such as developer opinions, standard best practices, test suites & source codes [18, 19, 1, 20].

The primary phase of this research is gathering or generating a considerable number of bugs to analyze. The studies show that it is tough to isolate, distinguish and regenerate the bugs in real projects for evaluation. For that reason, the usual practices depend on the manual seeded bus or mutation testing techniques [21]. Each of these techniques has its limitations. The manually seeding can be based on the expectations, understanding, and experience of the person, and also it is a time-consuming task. On the other hand, mutation testing is an expensive process compared to manual seeding. The mutation technique changes the small portion of the code and checks if the tests recognize errors. The researchers have shown that the mutation technique is much more powerful and capable of subsuming [22] than other structural testing techniques such as Data flow and Control flow testing.

2.2 Bug related studies

2.2.1 BUGSJS

In 2019 at the IEEE International Conference on Software Testing, Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah presented [19] a benchmark and classification for the Javascript bugs.

They have validated 453 real bugs from 10 popular JS frameworks such as Bower, Eslint & Express. The reason for selecting popular projects as those projects have considerable community support and usually follow standard best practices, including bug reporting and tracking.

They queried using GitHub API to capture the closed issues with linked bug-fixing commits and assigned specific labels. They have excluded rejected pull requests and bugs linked with more than one bug fix because they assumed the first-time attempt was unsuccessful. In addition, those bugs and fixes should adhere to reproducibility and Isolation. As you can see in the Figure 2.1, After the bug collections step, they manually validate fixed bugs to have unit tests and evaluate the unit test involvement in the fixes. The processes the list of bug candidates.

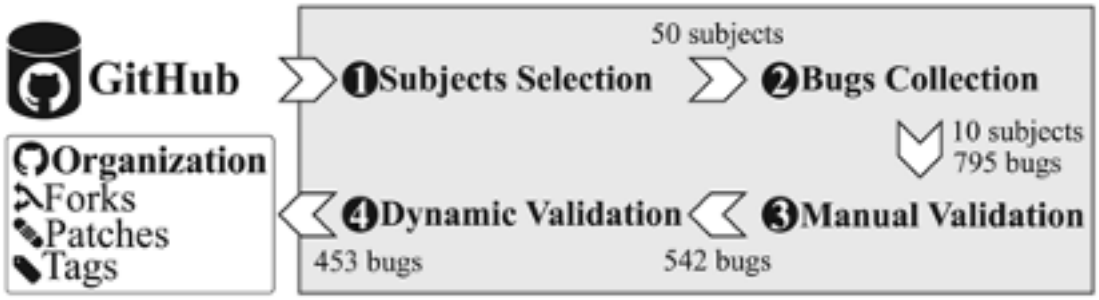


Figure 2.1 Overview of the bug selection and inclusion process. Adapted from [19]

Rule name	Description
Isolation	The bug-fixing changes must fix only one (1) bug (i.e., must close exactly one (1) issue).
Complexity	The bug-fixing changes should involve a limited number of files (≤3), lines of code (≤50) and be understandable within a reasonable amount of time (max 5 min).
Dependency	If a fix involves introducing a new dependency (e.g., a library), there must also exist production code changes and new test cases added in the same commit.
Relevant changes	The bug-fixing changes must involve only changes in the production code that aim at fixing the bug (whitespace and comments are allowed).
Refactoring	The bug-fixing changes must not involve refactoring of the production code.

Figure 2.2 Bug-fixing commits inclusion criteria. Adapted from [19]

During the Manual Evaluation method, they excuse some commits to simplify the evaluation shown in the Figure 2.2.

Apart from that, they label the commits manually by four reviewers individually. Then they take the final decisions for the label in the meeting. At the high level, they have categorized bugs into four categories.

1. Incomplete feature implementation
    a. Incomplete data processing (27 issues) - Missing type check, empty input parameter, missing null checks, handling of space, handling social, chars.
    b. Missing input validation (72 issues) - Missing null check, missing handling of spaces, missing handling of special chars.
    c. Error handling (16 issues)
    d. Incomplete config processing (16 issues) - Missing type check
    e. Incomplete output message (3 issues)
2. Incorrect feature implementation (162 issues)
    a. Incorrect data processing (64 issues) - incorrect type and initial value comparison
    b. Incorrect input validation (61 issues) - Unnecessary type check, incorrect handling of special chars, empty input parameters
    c. Incorrect file path (12 issues)
    d. Incorrect output (5 issues)
    e. Incorrect configuration processing (9 issues)
    f. Incorrect handling (7 issues)
    g. Performance (4 issues)
3. Generic programming errors (29 issues)
    a. Typo (3 issues)
    b. Return statement (4 issues)
    c. Variable initialising (6 issues)
    d. Data processing (3 issues)
    e. Missing type conversation (3 issues)
    f. Loop statement (1 issue)
4. perfective maintenance (7 issues)

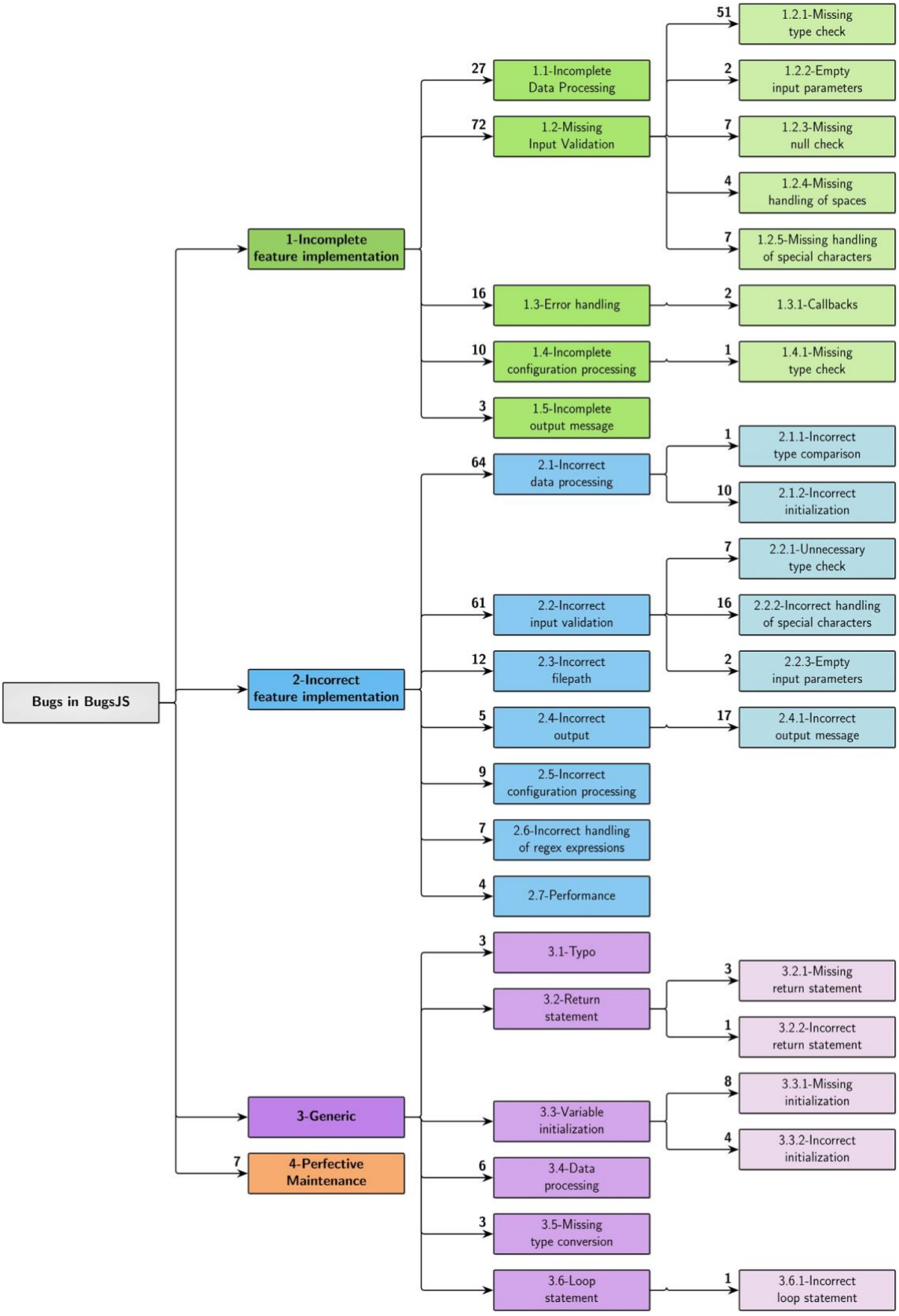


Figure 2.3 benchmark of Javascript programs of bugs. Adapted from [19]

This research has reviewed many papers and provided how they differ from other researchers even though most of the classification is done manually. During their manual classification, they filtered all the edge cases' ambiguities to avoid the complexity of the result. According to this paper, Mutation Testing is practical to analyze Javascript because of its dynamic nature and agility. Since researchers label individually and at the end of the day, those four researchers have to agree on the final classification label, providing consistency and accuracy.

2.2.2 To type or not to type

In 2017, Z. Gao, C. Bird, and E. T. Barr presented Javascript analysis on static type-related issues under the IEEE software [6]. Their focus is to compare the impact of static type on Javascript projects. The authors have considered a popular type system called Typescript from Microsoft and Flow from Facebook. They have studied 398 projects with over 7 million lines of code. However, they studied only 400 bug reports.

At the high level, their approach is to find the bug fix commits in production, then go back and add type annotation and check whether static typing can catch the fault. Their experiment mentioned a few limitations, such as manual fixing might fail to resolve the bug, and bug triggering may exist outside the region touched by the code. Also adding type checks may identify unrelated faults that may not be attached to the bug fix.

Although the authors do not have fully typed versions of the projects, they only had to add type annotation for the lexical scope. To maintain consistency, they have not used full formed annotations. For example, if there is a calculation to get the sum of two different numbers (var a=b+c), the output should be number (a: number =b+c) instead of boolean (a: boolean =b+c).

$$\frac{\left|\{b_i \in B \mid \neg tc(a(b_i))\}\right|}{|B|}$$

Figure 2.4 Formular. Adapted from [6]

The Figure 2.5 depicts how they classified ts-detectable and not detectable (hashed partition) bugs. However, the systems (Specifically the research, its Typescript, and Flow) cannot identify all kinds of fixed public bugs. However, Flow and TypeScript can detect type mismatches, prevent type mismatches, coercions (normally hidden by Javascript), and undefined properties and methods.

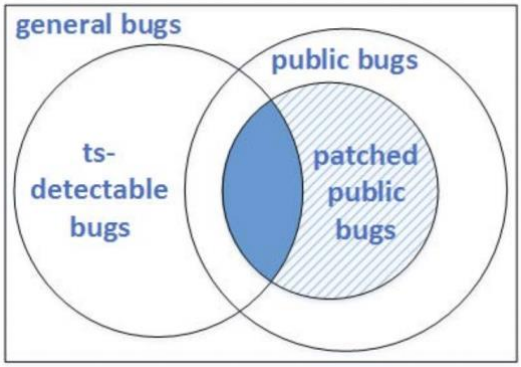


Figure 2.5 TS detectable bug chart. Adapted from [6]

The Figure 2.6 shows code blocks on how the authors annotate buggy programs with type systems to detect issues in the development stage. The code block (b) shows how usually the sum of two numbers function works. If we put two numbers, it can produce the correct output.

Nevertheless, if we put string and number, as example-3, number and 0 as string, the result becomes 30 instead of 3 (code block a). If the code block has a type system, it accepts only the numbers. Otherwise, will show errors or warnings to the developer at compile time or on IDE.

```
1  function addNumbers(x, y) {
2      return x + y;
3  }
4  console.log(addNumbers(3, "0"));
```

(a) The buggy program.

```
1  function addNumbers(x, y) {
2      return x + y;
3  }
4  console.log(addNumbers(3, 0));
```

(b) The fixed program.

```
1  function addNumbers(x:number, y:number) {
2      return x + y;
3  }
4  console.log(addNumbers(3, "0"));
```

(c) The annotated, buggy program.

Figure 2.6 Typescript annotation code. Adapted from [6]

When it comes to finding the correct sample size for the accuracy of the research, they considered standard sample size computation; they needed 400 bugs to achieve a confidence level of 95% and a confidence interval of 5%. Also, they have used open coding for bug categorization, a qualitative approach for categorizing observations. The Figure 2.7 diagram shows the taxonomy of the undetectable bugs from Flow and TypeScript.

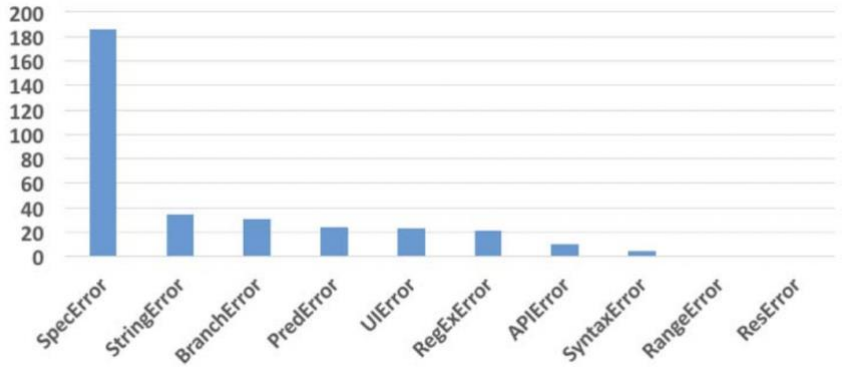


Figure 2.7 Undetectable bugs from Flow and TypeScript. Adapted from [6]

Based on the research, they have recognized that Typescript and flow can identify 15% of bugs at the development time. Since developers are well paid, the 15% is a considerable number of bugs because bug identification to bugfix release takes a significant process. However, the number of percentages can vary depending on the project, such as open-source or commercial projects, the experience of development

teams, quality assurance process, code quality checks, coding standards & requirement specifications. However, this research has used the most popular open-source projects like jQuery and Nodejs, but most real-world projects are not related to building frameworks. So, in that case, the impact may differ depending on the project.

2.2.3 Qualitative methods in empirical studies of software engineering

Many studies quantify the bugs in software engineering because the software needs to fix bugs and build bug-free solutions. When it comes to the bugs and engineering complexities, those are dependent on many factors such as [23] technical issues, human behavior, and the intersection of machines and people.

There are practical and theoretical ways to research software engineering. In this research, we are more focused on empirical studies because bug detection is more related to human behavior, observation, and developer experience. According to C. B. Seaman [23], there are two methods to capture empirical data: Participant Observation, Interviewing, and Coding. We will not consider the interview because it is not a part of our study. However, participation observation does not mean just involving humans. It means gathering information by reviewing the code, notes, videos, audio, or files. According to them, the real data recorded, such as comments, commits, and bug reports, are useful sources for the research.

In studies, we have qualitative and quantitative data, which means qualitative data is represented as words or pictures, and quantitative information is represented as numbers. To combine qualitative and quantitative data, researchers extract values for quantitative variables from qualitative data to build statistical analysis, called coding [23]. An example of coding is if we give a C++ class with some logic and ask people to rate the complexity, they will provide different complexities depending on their experience. In that case, we need to narrow down the questions and ask specific questions. Then they will give similar answers. Learning from this is that I need to narrow down the particular research area to get good coding results.

As mentioned in previous studies, those projects also use empirical studies to capture the data because most bug-related studies need human interactions. It does not mean that we cannot automate it because many studies are automated [10]. However, the

author says that Qualitative is richer than quantitative because it contains diversified data that increase confidence and multiple analyses. However, after applying all the filtering, they could identify only 80 bugs that can be fixable by Typescript or flow.

2.2.4 Discovering bug patterns in JavaScript

Q. Hanam, F. S. de M. Brito, and A. Mesbah presented a paper called "Discovering bug patterns in JavaScript" [17], explaining that most bug-finding tools discover bug patterns using best practices, developer experience, and anecdotal observation. However, it does not provide a solution for the frequent bugs that occur in practice which are essential to fix. So they introduced a solution called BugAID [17] a semi-automatic technique to find the most noticeable and widespread bug pattern using unsupervised machine learning using language construct-based changes filtered from Abstract Syntax Tree (AST) differencing of bug fixes in the code base. Since they used a semi-automatic technique, they could do a large-scale study by mining 105k commits from 134 Javascript projects (Server Side) of common bug patterns.

The authors say that currently, Javascript lacks IDE support [24] for analyzing and providing warnings about the bugs in their code. Although, those alters very common in strongly typed languages such as Java because compilers and tools like FindBugs [25] can provide better errors and warnings. Also, Javascript IDE lacks support because of the unique behavior, such as higher-order functions, dynamic code evaluation, event-driven, asynchronous flow, and weak typing. Their goal was

1. Build a systematic semi-automation tool to discover bug pattern
2. Discover common frequent bugs in Server-Side Javascript

They have followed the following approaches to discover frequently occurring bug patterns in Javascript with public GitHub projects.

1. Reducing the search space

   One way to find bug patterns is to check commit history one by one, but it might take humans several days for a project with thousands of commits. Since it is looking for common bug patterns, it would not be feasible. So, they reduce the search scope by only focusing on commit changes instead of the bug report and selecting commits with 1 to 6 statement changes. This approach is an

important decision to scale unnecessary filter commits and find high-quality data sets.

2. Grouping Cross-Project Bug Patterns

   Since they had no prior knowledge about the bug pattern grouping, they decided to perform cluster analysis using machine learning. Since selecting the best feature vector and clustering algorithm are challenging, they line level differences or AST differences on the source code to avoid noise such as variable names and minor differences. In their early discovery, that approach did not consider the semantics of the changes, So as a better approach, they used "change classification." However, one of the primary purposes was to find unknown change types in this classification, but the classification (called "basic change types" - BCTs) must be manually specified. So it does not scale to capture new patterns with additional data.

3. Learning Change Types

   They decided to automatically learn BCTs by combining information from a mode of languages by the following abstraction for capturing BCTs with the following components

   1. **Language construct type**: method, field names, statement types, operators, behavior (ex: casting), and conde conventions
   2. **Language construct context**: Ex: reserve words such as this can occur in many contexts
   3. **Modification type:** How the commit modified language construct.
   4. **Name:** assigned value for language construct

4. Language Contract Selection

   Their goal was to discover patterns that are inherent to Javascript. Therefore they used reserver words in Javascript and considered the following behaviors.

   1. falsey: In Javascript, all variables can be cast to boolean. Such as undefined, null, NaN, and 0, false means false. Otherwise, it is true.
   2. typeof - find the type of variable

3. callback - Javascript heavily depends on the callback
4. error. - ex: error-first callback is a pattern in Javascript

We think that those are very critical and unique things to consider in Javascript

5. Extracting Basic Change Types

The Figure 2.8 shows the summary of the extracting BCT function. As it shows, it takes the subject program and returns the commits and change types database. It provides a set of [F:= {fb1, fr1}...{fbn, frn} of n] buggy and repaired files for each commit in the commit history. Then each pair of F, the code will compute BCT using AST (abstract syntax tree) differencing to get more accurate differencing than line level differencing, and it also computes find-granularity changes for learning BCT by each node in AST. Likewise, it follows that method, and at the end, it will store each c (feature vector) in the relational database to query the data in the future.

Algorithm 1: Basic Change Type Extraction

Input: $\mathbb{P}$ (subject programs)
Output: $\mathbb{D}$ (database of commits and change-types)
$\mathbb{D} \leftarrow \emptyset$
foreach $p \in \mathbb{P}$ **do**
  $\mathbb{C} \leftarrow \text{Commits}(p)$;
  foreach $c \in \mathbb{C}$ **do**
    $\mathbb{T} \leftarrow \emptyset$;
    $\mathbb{F} \leftarrow \text{ModifiedFiles}(c)$;
    foreach $\{f_b, f_r\} \in \mathbb{F}$ **do**
      $\{\text{AST}_b, \text{AST}_r\} \leftarrow \text{ASTDiff}(f_b, f_r)$;
      $\mathbb{M} \leftarrow \text{ModifiedFunctions}(\text{AST}_b, \text{AST}_r)$;
      foreach $\{m_b, m_r\} \in \mathbb{M}$ **do**
        $\mathbb{T} \leftarrow \mathbb{T} \cup \text{ExtractBasicChangeTypes}(m_b, m_r)$;
      end
    end
    $\mathbb{D} \leftarrow \mathbb{D} \cup \langle c, \mathbb{T} \rangle$;
  end
end

Figure 2.8 Basic Change Type Extraction. Adapted from [17]

Once they store the data, they filter candidate commits by applying the following filters

1. Statement change between 1 to 6
2. Not contain merge in the commit message
3. Contains at least one BCT that does not have language construct context "S"
4. Contains at least one BCT that does not have modification type' U

   Since each commit in the query results has been converted into an instance of a feature vector, those vectors include the number of modified statements and a list of words.

6. Clustering and ranking

   As they got a bag of change types with numbers that happens commonly. They used the DBSCAN clustering algorithm, which groups feature vectors close together (density-based). They also use the Manhattan distance function to calculate the shorter distance between commits with the same BCT. It provides them to identify frequently occurring bug patterns with change types across the projects.

7. Change Type Inspection

   The shows how they collected feature vectors for the three commits. In this stage, they need to inspect manually, and to speed the process, they considered only change types with more than five commits and represented on multiple projects.

Table 2.1 Chante Type Inspection. Adapted from [17]

Proj.	Commit	Modified Statements	Change-type Header
async	63b	1	B_C_I_falsey [1] B_C_R_falsey [0] M_MC_I_bind [0]
bower	2d1	4	B_C_I_falsey [1] B_C_R_falsey [2] M_MC_I_bind [0]
express	5c4	2	B_C_I_falsey [0] B_C_R_falsey [0] M_MC_I_bind [1]

First, they take samples randomly by each change type. If the change type contains 100 commits from 40 projects, the sample size will be 40 because the project's purpose is a cross-project bug pattern. Then the inspectors spend time understanding the change with the commit message, code change, and description. If those descriptions match the code changes, they keep the change type as it is. Otherwise, introduce a new group.

8. Implementation

   They build their tool called BUGAID with the following technologies.

   1. Mozilla Rhino [28] - JavaScript ASTs, Runtime environment
   2. GumTree [29] - AST differencing
   3. Clustering [30] - Weka

2.2.4.1 Pervasive bug patterns

As a result of this research, they have identified 219 bug fixing change types and explained 13 pervasive bug patterns that happen across multiple projects [17]. They think that IDE can prevent those bugs with better plugin support. The following shows pervasive bug patterns and solutions.

1. Dereferenced Non-Values

This means that Javascript can throw an error called TypeError when trying to use null or undefined values. They provide three repair solutions.

1. Protect with false check: if values are falsy, then not to proceed in the next line

```
+ if (!target.parts) return;
    part = target.parts[playingPart];
```

Figure 2.9 Protect with false check. Adapted from [17]

2. Protect with no value check

```
- if ( val <= 0 ) {
+ if ( val <= 0 || val == null ) {
    val = curCSS( elem, name );
```

Figure 2.10 Protect with no value check.. Adapted from [17]

3. Protect with type check: Some type has unique methods. For example, if the variable type is a string, it has the toLowerCase method, but numbers do not. If someone tries to access toLowerCase from a number it will break.

```
+ if (typeof torLink === 'string') {
    if (torLink.toLowerCase().match(...) != null) {
```

Figure 2.11 Protect with type check. Adapted from [17]

2. Incorrect Comparison

Since there are many ways to compare, such as falsy values, check by value, or check by both value and type in the Javascript. They suggest using a strict value comparison like this.

```
- if (typeof opt.default != 'undefined')
+ if (typeof opt.default !== 'undefined')
    self.default(key, opt.default);
```

Figure 2.12 Incorrect Comparison – I. Adapted from [17]

```
- if (encoded_test === "TEST")
+ if (encoded_test == "TEST")
```

Figure 2.13 Incorrect Comparison – II. Adapted From [17]

3. Missing argument

Javascript function parameters do not require passing all the parameters in the function definition. So, they suggest passing null for missing parameters to avoid ordering issues.

```
1 if (completed === arr.length) {
2 -   callback();
3 +   callback(null);
```

Figure 2.14 Missing argument. Adapted from [17]

4. Incorrect API configs

They do not have the correct solution for this configuration because Javascript is a dynamically typed language. The form of configuration can be JSON or Object. If they use Java, they can easily find it in compile-time/ or in IDE because it is a statically typed language.

```
1   grunt.initConfig({
2     clean: {
3 +     force: true,
4       build: ['dist']
```

Figure 2.15 Incorrect API Config. Adapted from [17]

5. "this" not correctly bound

The most problematic reserved keyword in Javascript is "this". Most of the time, developers incorrectly assume that "this" is bound to function definitions. So they suggest using a local variable to assign "this" (ex: var self = this).

```
1 + var self = this;
2   self.serverConfig.connect(function(err, result) {
3     self._state = 'connected';
```

Figure 2.16 "this" not correctly bound. Adapted from [17]

6. Unhandled exception

Javascript throws many different exceptions, such as asynchronous, event-based, and type errors. They suggest following callback error styling and try-catch blocks.

```
1 + try {
2     html = kiwi.jade.compile(str)(...);
3 + } catch (e) {
4 +   response.statusCode = 500;
```

Figure 2.17 Try Caught. Adapted from [17]

```
1|   Server.prototype.handle = function(outerNext) {
2|     function next(err) {
3| -     outerNext();
4| +     outerNext(err);
```

Figure 2.18 Callback error. Adapted from [17]

```
1|   return fs.readFile(path, function(err, buffer) {
2| +   if (err) {
3| +     throw err;
```

Figure 2.19 Callback error exception. Adapted from [17]

2.2.4.2 Conclusion

Unlike other solutions described in the literature review, this report provides a detailed solution for automating bug classification and gathering a large amount of data. They have mined 105k commits from 134 Javascript projects which is a solid sample size.

However, they suggested solutions for the bugs that can be easily identified if they use type language. We think that if they use Typescript in their project, they could reduce most of the issues.

These findings can be helpful for the IDE, extensions, and plugin developers to detect common bug patterns in the Javascript, so developers to avoid making common mistakes and save time for bug fixing.

2.2.5 To Type or Not to Type - A Systematic Comparison

Bogner, Justus & Merkel, Manuel published a paper called "To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub" [26] in 2022. The authors state that although Typescript solves type-safe issues and provides better software quality than Javascript applications, there is no sufficient empirical evidence for that. Therefore they mined 604 Github projects (209 for Javascript and 305 for Typescript) with over 16 million lines of code (LoC) using GitHub API and SonarQueue. They analyzed the following software qualities

1. Code Quality (# of code smells per LoC)
2. Code understandability (cognitive complexity per LoC)
3. Bug proneness (bug fix commit ratio)

4. Bug resolution time (mean time a bug issue is open)
5. For Typescript - Frequency of ignoring type-safety using any

Based on the above qualities and the other research papers, they defined eight hypotheses for statistical analysis under two main research questions.

1. RQ-1: Do TypeScript applications exhibit better software quality than JavaScript applications?
2. RQ-2: RQ2 Do TypeScript applications less frequently use any type exhibit better software quality?

Table 2.2 Null hypotheses with their alternatives for both RQs. Adapted from [26]

RQ	Metric	Null Hypothesis	Alternative Hypothesis
RQ1	Code smells per LoC	H_0^1: TS applications exhibit less or equal code quality than JS applications.	H_1^1: TS applications exhibit better code quality than JS applications.
	Cognitive complexity per LoC	H_0^2: TS applications exhibit less or equal code understandability than JS applications.	H_1^2: TS applications exhibit better code understandability than JS applications.
	Bug fix commit ratio	H_0^3: TS applications are more or equally prone to bugs than JS applications.	H_1^3: TS applications are less prone to bugs than JS applications.
	Bug resolution time	H_0^4: TS applications require more or equal bug resolution time than JS applications.	H_1^4: TS applications require less bug resolution time than JS applications.
RQ2	Code smells per LoC	H_0^5: The any type frequency correlates not or positively with code quality in TS applications.	H_1^5: The any type frequency correlates negatively with code quality in TS applications.
	Cognitive complexity per LoC	H_0^6: The any type frequency correlates not or positively with code understandability in TS applications.	H_1^6: The any type frequency correlates negatively with code understandability in TS applications.
	Bug fix commit ratio	H_0^7: The any type frequency correlates not or negatively with bug proneness in TS applications.	H_1^7: The any type frequency correlates positively with bug proneness in TS applications.
	Bug resolution time	H_0^8: The any type frequency correlates not or negatively with bug resolution time in TS applications.	H_1^8: The any type frequency correlates positively with bug resolution time in TS applications.

They created a python script to collect necessary data from the GitHub API, but they face a significant limitation using the GitHub API with an access token that can only send 5,000 requests per hour. They considered only the repositories with more than five stars, excluded forked projects, and ignored projects with Frameworks like Angular. Also, they selected projects that were created between 2012 to mid-2021. One of the reasons to select 2012 is because Typescript was introduced in 2012.

When it comes to the sample sizing, they combined a confidence level of 95% and a level of precision of 5%. After applying above mentioned filtering, the authors found 300,00 Javascript and 60,000 Typescript projects. However, most include databases, engines, plugins, extensions, templates, build tools, CLI (Command Line Tools), and Utilities. Therefore, they hypothesized population size as 1/20(approximately) of that.

Using that calculation, they need 375 Js and 341 Typescript projects. After that, they sorted all of them by stars, selected batches of 300 projects, and analyzed them by the following measures.

1. Project is an application: verified this using the README.md file and project description using Latent Dirichlet Allocation (LDA) topic modeling algorithm. It can detect keywords with wight and exclude like "plugin", "module", "extension", "API", "database", "framework", "library"
2. The project has closed bugs: The closed bugs are needed to calculate bug resolutions. They also selected only English language projects.
3. The project has more than 30 commits: To ensure reasonable development activity.

Although the above process is automated, they manually checked all the passed projects. As a result, their final sample size is 299 JS and 305 TS repositories. The Table 2.3 shows the statistics.

Table 2.3 Properties for JavaScript (JS) and TypeScript (TS). Adapted from [26]

	Projects	Total LoC	Total Commits	Total Issues
JS	299	7,496,726	235,535	56,578
TS	305	8,683,600	340,164	157,497
Total	604	16,180,326	575,699	214,075

Table 2.4 Share of the primary programming language (PL). Adapted from [26]

	PL ≥ 90%	90%> PL ≥ 80%	80%> PL ≥ 70%	70%> PL ≥ 60%
JS	125	84	49	41
TS	138	82	45	40

They analyzed those projects using the SonarQube, a tool to analyze code smells and quality. To work with SonarQube, they had to download all the project locally and run it. One of the challenges with Typescript is that it depends on the Typescript configuration. To maintain consistency, they deleted the “.tsconfig” and replaced it with the default file. Also, they extracted "nclock" from SonarQube to remove empty

lines and comments. While considering bug resolution time, they excluded issues that open and closed within 5 minutes; issues are opened for more than one year. After applying those filters, the projects with less than five bug issues were not included for metrics. To get all any type usage per Typescript project, they used eslint with the "Typescript-eslint" plugin, which has a "no-explicit-any" rule. Again, they automated this process by deleting the existing eslint file and replace with the default config using python script.

1. Code smells

   They discovered that Typescript has relatively more minor code smells per lot. The Table 2.5 shows that Javascript has more than twice as many codes smells as the Typescript. It says the Javascript application has 12 code smells per 1000 LoC, more than the Typescript application.

Table 2.5 Code smells for JavaScript. Adapted from [26]

	Projects	LoC	Code Smells	Mean	Median
JS	299	7,496,726	217,957	0.025545	0.020132
TS	305	8,683,600	95,132	0.013368	0.011143

2. Cognitive complexity

   The report shows that Javascript code complexity is five times higher than Typescript. Low complexity means better code understandability. The comparison shows 90 (TS) vs. 332 (JS) per 1000 LoC code complexity.

Table 2.6 Cognitive complexity. Adapted from [26]

	Projects	LoC	CC	Mean	Median
JS	299	7,496,726	3,435,760	0.321999	0.157013
TS	305	8,683,600	722,687	0.089552	0.077416

3. Bug fix (BF) commits

   The Javascript apps have roughly one bug fix for every eight commits, and Typescript has one bug fix for every five. It shows that Typescript has a higher fix commit ratio than the Javascript applications.

Table 2.7 Cognitive complexity. Adapted from [26]

	Projects	Commits	BF Commits	Mean	Median
JS	285	235,535	30,717	0.126364	0.109375
TS	288	340,164	66,488	0.206436	0.163461

4. Bug resolution (BR)

   At this time, their hypothesis fails that Typescript because the numbers show that Typescript needs more time than Javascript to fix a bug. On average, fixing a bug in Typescript takes an additional day.

Table 2.8 Bug resolution. Adapted from [26]

	Projects	Bug Issues	Mean BR Time	Median BR Time
JS	183	15,894	31.86 days	25.59 days
TS	245	51,817	33.04 days	28.49 days

5. Effect of the any in TS

   Since they added eslint to capture any in Typescript, they found 79,735 any type violations over 305 Typescript projects, on average 261 per project. After normalizing, it shows that the TS apps use any for every 100 lines of code.

Table 2.9 Descriptive statistics on any type usage. Adapted from [26]

	Min	Max	Mean	Median	Total
any types	0	5326	261.43	70	79,735
any types per LoC	0	0.1063	0.0103	0.0072	–

To analyze the impact, they had to use a scatter plot. The frequency of "any" type shows a positive impact but not much. The chart shows the same results for fix commit ratio and cognitive complexity. There is no significant improvement by using less "any" in the code base. They assumed that most of the Typescript might be more frequently chosen by experienced developers, and they usually make less code smells and complex code. Also, experience developers use static analysis like eslint to remove code smells frequently. They also suspect that TS apps average 2.5k stars (759,619 in total), while JS apps averaged only 604 (180,690 in total). Their theory is that when a project becomes popular, more senior developers are willing to contribute, and it helps to increase the code quality.

The Table 2.10 shows the result of the initial hypothesis testing. The research clearly shows that Typescript increases the code quality and understandability. Based on that, the authors suggest that Javascript developers be mindful of the code quality. However, it does not mean that Typescript does not provide outstanding support to prevent the bugs. In the authors' words, they suggest that Typescript developers should not try it religiously.

Table 2.10 Summary of hypothesis testing results

RQ	Alternative Hypothesis	Test Statistic (U)	p-Value	Cohen's d or ρ	Accepted
RQ1	H_1^1: TS applications exhibit better code quality than JS applications.	28842.5	**2.78e-15**	0.671	Yes
	H_1^2: TS applications exhibit better code understandability than JS applications.	27219.0	**5.14e-18**	0.339	Yes
	H_1^3: TS applications are less prone to bugs than JS applications.	54395.5	0.99	–	No
	H_1^4: TS applications require less bug resolution time than JS applications.	23281.0	0.75	–	No
RQ2	H_1^5: The any type frequency correlates negatively with code quality in TS applications.	–	**2.48e-06**	0.26	Yes
	H_1^6: The any type frequency correlates negatively with code understandability in TS applications.	–	**4.7e-04**	0.19	Yes
	H_1^7: The any type frequency correlates positively with bug proneness in TS applications.	–	0.76	-0.04	No
	H_1^8: The any type frequency correlates positively with bug resolution time in TS applications.	–	**0.0034**	0.17	Yes

The output of this research paper says that TS applications have remarkably better quality and understandability than Javascript applications. However, the bug fix commit ratio for TS projects was more than 60% higher than Javascript. On average, Typescript projects need an additional one day to fix a bug. Also, using less any keyword in Typescript shows more benefits in all the metrics except bug proneness. The authors say that while using Typescript provides better code quality, it does not automatically help to improve the code quality.

2.3 Bug collections & classifications techniques

Collecting bugs and classification is time-consuming and challenging to collect bugs effectively [27, 28]. Therefore many researchers studied how to collect and classify bugs to achieve the research outputs. Most of the studies related to software engineering depend on the bugs classification, and most bugs reports are misclassified between bugs and non-bugs [27]. This section will explain how several researchers approach this problem with different techniques.

Ocariza et al. have presented an analysis and classification for Javascript bugs to understand the root cause [29]. They have invented a tool called "HOLOCRON" to detect bugs using an anomaly-based inconsistency approach. In that research, they have studied 509 bugs with our 2 million lines of code. They have limited the scope of their research into MVC frameworks (AngularJS, BackboneJS, and Ember.js) because of the rising popularity in 2017 and not interacting directly with the DOM. Most of the other studies required gathering bugs manually, but this paper has an automated solution for detecting issues. Using the codebase, they generate both HTML DOM and JS DOM trees and incorporate them into another tree called CodeTree. They have used a technique to find the anomalies using an algorithm that finds a similar code block and tries to identify anomalies in the code with the help of code smell techniques.

As discussed in the bug-related studies, a tool called BugAID [17] uses language construct-based AST to differencing bug fixes in the code base to identify bug patterns using an unsupervised machine learning approach, a semi-automated solution to identify bug patterns. First, they selected bugs in GitHub using several filtering mechanisms, such as selecting commits with only 1 to 6 statements and ignoring white and empty spaces on commits. Their primary purpose was to find bug patterns across projects. So, they built a classification called BCT. Using that, they could define the list of categories such as type Error Undefined, Type Error Falsely, Error Handling & Incorrect Comparison. Then they manually reviewed them randomly and grouped them into different categories if necessary. Using their techniques, they could find 219 categories by mining 105k commits over 134 Javascript projects. However, this can be an issue for some studies because 219 is a considerable number, and it needs another classification to classify those into small numbers around 20-30.

As mentioned earlier, Gao et al. [6] represent an analysis of Typescript and flow over Javascript, which says that using a type-system can reduce 15% of the bugs. Their experiment setup is similar to Le Goues et al. [30] research, which aimed to identify real-world historical bugs sampled from Github Project and determine what portation would be fixable by automatically using the program. Likewise, Gao et al. [6] identify bugs in real-world Javascript projects and convert them to Typescript and Flow, then

analyze how it detects the issues. First, they needed to capture many bugs with bug reports for sampling. They manually selected eight Javascript projects based on project size, popularity, number of contributors, and using Node.js and jQuery. The next step was to identify the candidate fixes. Instead of going through the closed pull requests, they analyzed from the SHA-Commit and found the relevant pull request. Then they manually assessed whether the commit intended to fix the issue rather than code refactoring, enhancement, or another improvement. Then they followed a technique called open-coding to categorize observations. The researchers group them into several groups: EvalError, RangeError, URIError, and SyntaxError. However, they need 384 bugs based on their level and confidence interval of 95% and 5%. This categorizing approach is very similar to what we need to achieve in this report.

Wang et al. [31] have researched NodeJS concurrency bugs, including bug patterns and root causes, impacts, manifestation, and fix strategies on 57 issues in 53 Node.js applications. Since there were not enough, especially concurrency bugs in single or multiple projects, they had to pick bugs from all the GitHub repositories using GitHub advanced GitHub filtering and keywords such as concurrent, race, synchronization, and deadlock. They have searched 97,000 NodeJS projects and found 1,583 bug reports. After manually validating those bugs, they picked 214 concurrency bug reports in 147 projects. They created four questions

1. Bug patterns and root causes
2. Bug impacts

3 Bug manifestation

4. but fix strategery

to get answers to those questions from each bug. Then they could only pick 57 bugs that could provide a reasonable answer for those questions.

They selected categories based on the taxonomy in related work [32, 31]. They studied each bug and assigned it to different categories. Then they studied all the categorized bugs and decided to adjust the category based on the nature of the bugs, such as renaming the category, merging the category, or removing the useless category. In that

process, they decided that at least two authors should review. If they see any conflict, they reinvestigate until they form a final decision.

According to those studies, there are different ways to collect bugs, such as manual, automated, and hybrid. The analysis and categorizing depend on the purpose of the project and the nature of the project. For example, although the research searched all over Github, they could find only 1583 concurrency bugs. It was not straightforward for them to automate the categorizing processes. However, BugAID [17] tool could use the automated solution to capture and categorize bugs because the purpose is to find general bugs categorization. In our case, we are looking for specific research, and the automated solution will not work because we need to analyze the detectability manually.

2.4 Quantifying and analyzing detectable bugs

The studies show that various prediction models have been developed to increase the quality of software products by using bug characterizing [33]. In this section, we review available methods to analyze and quantify bugs.

Due to many reasons, the code becomes more uncertain and irregular. To quantify those behaviors, Kumari et. [33] suggests applied entropy-based measures for summary and comments submitted by users and entropy-based measures for identifying uncertainties, irregular patterns, and bug fixes.

The number of fixed bugs is one of the facts to measure the software reliability growth. The software reliability growth model curve follows exponential, S-shaped, or some mix of both. The models developed in the literature are based on either calendar time or testing effort function. The figure 30 shows how they measured the summary and comment entropy using the different models. As it shows, they collect the data by downloading bugs, summary descriptions, and comments on each bug. Then they apply Shannon's entropy calculation to calculate summary and comment entropy. Using these calculation results, they get the cumulative number of bugs monthly. Then they apply it to the different models and compare existing models.

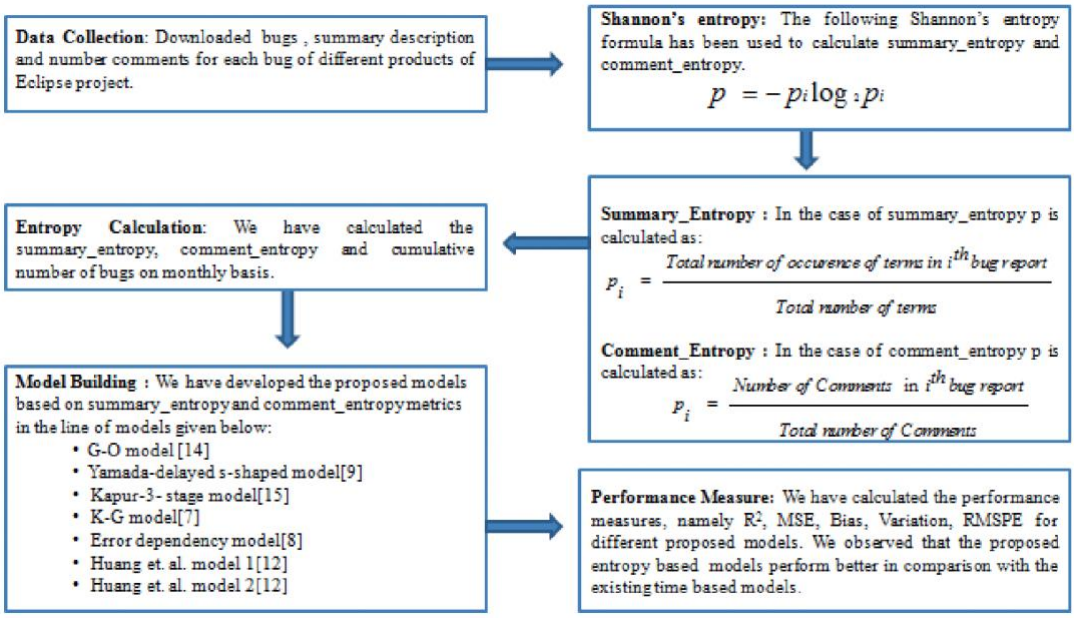


Figure 2.20 Block diagram of proposed methodology. Adapted from [33]

One of the primary objectives is to build high-quality software at a low cost. Since software repositories become large daily, that increases the uncertainty and a lot of noise. The author used Shannon's entropy [24] to calculate it as the measurement attribute. Random variables are defined by A = {a1, a2, a3, . . . , an}. Its probability distribution is defined by P = {p1, p2, p3, . . . , pn}, the random uncertainty defined by Sn as this;

$$Sn = -pi \log 2pi$$

In the case of summary_entropy, p is calculated as:

$$p_i = \frac{Total\ number\ of\ occurence\ of\ terms\ in\ i^{th}\ bug\ report}{Total\ number\ of\ terms}$$

In the case of comment_entropy, p is calculated as:

$$p_i = \frac{Number\ of\ comments\ in\ i^{th}\ bug\ report}{Total\ number\ of\ comments}$$

Figure 2.21 Entropy calculator for summary and comment. Adapted from [33]

As mentioned, they have used different mathematical models to calculate, such as G-O model, Yamada-delayed, S-shaped model, Kapur-3-stage model, K-G model and Error Dependency mode.

The Figure 2.22 formula shows the bug detection/fixing description where x(t) describes the cumulative number of bugs detected/fixed in a time interval [0,t]. The g(t) is the time-dependent fixing rate of bug per pending bug. SolvingEquation (1), we get the following at time t = 0, x(0) = 0.

$$\frac{dx(t)}{dt} \propto (y - x(t)),\ or\ \frac{dx(t)}{dt} = g(t)(y - x(t)) \qquad -(2)$$

$$x(t) = y\left(1 - \exp^{-\int_0^t g(t)dt}\right) \qquad -(3)$$

Figure 2.22 Formula I. Adapted from [33]

The Figure 2.23 shows how they applied to different models to quantify.

Model 1. G-O model [14]:

If $g(t) = g$, Equation (3) reduces to:

$$x(t) = y(1 - \exp(-gt)) \tag{4}$$

Model 2. Yamada-delayed S-shaped model [9]:

If $g(t) = \frac{g^2 t}{1+gt}$, Equation (3) reduces to:

$$x(t) = y(1 - (1 + gt)\exp(-gt)) \tag{5}$$

Model 3. Kapur-3-stage model [15]:

If $g(t) = \frac{g^3 t^2}{1+gt+\frac{g^2 t^2}{2}}$, Equation (3) reduces to:

$$x(t) = y\left(1 - (1 + gt + \frac{g^2 t^2}{2})\exp(-gt)\right) \tag{6}$$

Model 4. K-G model [7]:

If $g(t) = \frac{g}{(1+\beta\exp(-gt))}$, Equation(3) reduces to:

$$x(t) = y\frac{(1 - \exp(-gt))}{(1 + \beta\exp(-gt))} \tag{7}$$

Figure 2.23 Formula II. Adapted from [33]

3. METHODOLOGY

Microsoft introduced the Typescript in 2012, but still, not enough evidence proves the impact on "How impactful is it to use Typescript over Javascript to detect defects?"

As discussed in the literature review, we found that there are mainly two ways to design the methodology identity impact on bugs using Typescript. One approach is as Bogner et al. team suggest [26] we can collect both Typescript and Javascript projects from the Github, and analyze them using code smell solutions like SonarQube [34]. Then we can build the statistical analysis on how Typescript and Javascript impact defect detection.

The second method that Gyimesi et al. team; proposed [6] to collect javascript bugs from the public Github repositories and annotate those bugs into Typescript and check whether it can detect the bugs. This experimentation can be named the "What-if" style experiment, where we attempt to determine how many bugs could be prevented with typescript [6]. In our research, we decided to follow the second approach because we need to study the preventable bugs instead of code smells for the project.

3.1 Selecting projects and configuration

For this empirical study, the first step is to collect suitable public repositories. As described in most studies, they suggest collecting suitable public projects with bug reports [17, 19, 22]. In this step, our main objective is to find projects with qualified Javascript bugs to construct this study. However, actual bugs are hard to isolate, reproduce and characterize [19, 22]

As discussed in the literature review, the studies have selected projects based on different criteria. The BUGSJS [19] selected 10 popular JS frameworks such as Bower, Eslint & Express. In their selection criteria, they considered the popular and trending javascript projects because those projects often have large communities of developers. Most of the time, they follow best practices, including bug reporting and tracking. They measured the popularity of projects with higher than 100 stars and more than 200 commits since 2017. Also, they considered projects with labeling and testing included because it is helpful to identify the bug commits. Z. Gao et al. [6] selected 8 Javascript

projects using a set of criteria, including project size, popularity, number of contributors, and use of NodeJs and Jquery.

Based on those researches, we also decided to select popular and trending Javascript projects (minimum 50k stars on GitHub) because we needed active projects. As UGSJS [19] mentioned, we can assume those projects follow best practices, including reporting and tracking. Since we need to get more bugs in limited projects, we decided to select projects with a minimum of 500 merged pull requests because we needed only 385 bugs from all the projects. So, on average, we needed only 77 bugs from each project (since we selected five projects).

In our study, we selected a limited number of projects because we needed to manually set up those projects and annotate each bug into typescript, which is time-consuming. Although other studies have selected 8 and 10 projects, we selected only five because we could capture enough bugs for our sample size. The Table 3.1 shows the list of repositories with language, purpose, total merged PRs and GitHub status (as of 2020-06-22).

Table 3.1 Selected projects

Project	Languages	Purpose	Repo Created	Total Merged PRs	GitHub Status

ChartJs	98.2% Javascript 1.5% Typescript	Generate Charts	2013-03	2,643	Stars - 57.3k Forks - 11.5k
Three-js	53.4% Javascript 42.7% HTML	Generate Web GL	2010-03	9,781	Stars - 83k Forks 32.1k
Create-react-app	98% Javascript	Create React Application	2016-07	1,934	Stars - 95.6k Forks - 24.8k
Express	100% Javascript	NodeJS Web Framework	2009-06	256	Stars - 57.4k Forks – 9.7K
Lodash	100% Javascript	JS Utility Library	2012-04	573	Stars – 53.5K Forks – 6.5k

Each of those projects are different and follows all the criteria that we decided based on the studies.

1. ChartJS - This is a chart library that started on 2013. This is the first choice when developers need to add Chart library into their website for free.
2. ThreeJS – One of the most popular libraries among the developers to create WebGL related development.
3. Create-React-App – React is the one of famous UI framework to develop web application which is developed by Facebook. The Facebook has assigned dedicated team to develop this.
4. Express – This is the popular API Framework for NodeJS. Purely written in Javascript.
5. Lodash – This is the utility framework for all kind of Javascript applications

After selecting projects, we needed to configure that project to support Typescript because we needed to annotate the bug into Typescript and check whether Typescript could detect the defects. In order to do that, we need to set up the Typescript Configuration file, which contains all the configuration information related to the Typescript rules. We can configure Typescript in different strict levels, but we decided to use the same configuration file as similar studies [26] for each project because of the consistency of the results. If we follow two different configurations, the results can be different. The Figure 3.1 shows the default Typescript configuration on the left side.

The right side shows how Typescript detects possible undefined. If we set “strictNullChecks=false” in the Typescript Configuration, it will not detect that undefined error. That is why using the same configuration in each project is essential. However, we had to change some minor configurations depending on the project, such as setting the root folder path.

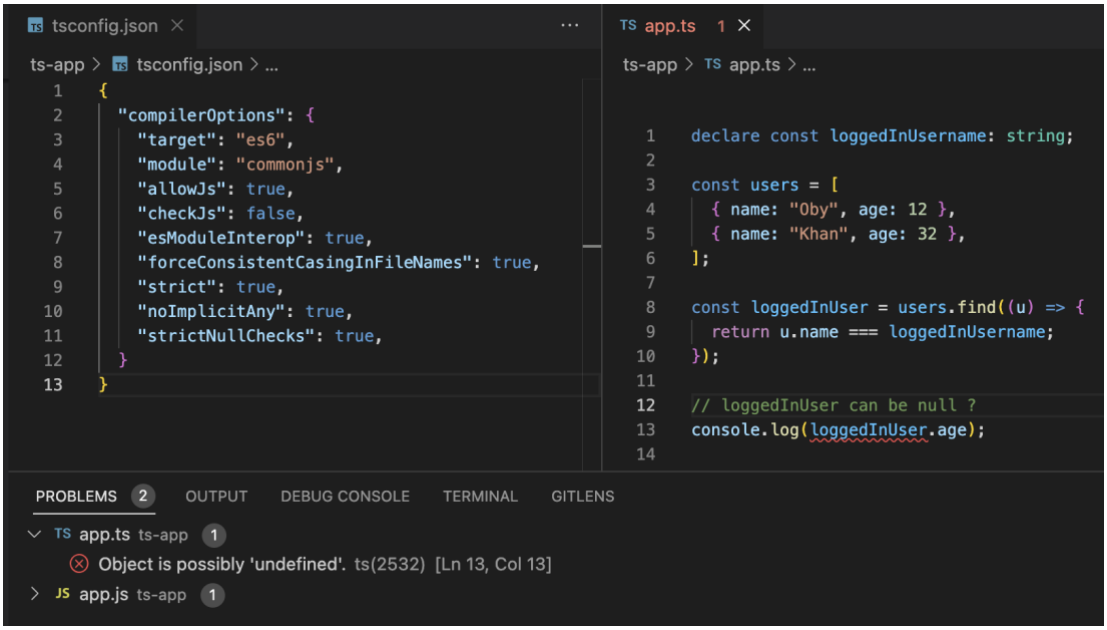


Figure 3.1 Typescript Default Configuration

3.2 Selecting candidate bugs

We need to collect sufficiently large to support statistical inference. Similar studies have used standard sample size calculation with a confidence level of 95% and a confidence interval of 5% [6, 35]. Based on that configuration, the Bongner et al. [26] selected 341 projects as sample size, Z. Gao et al. [6] selected 384 bug commits over eight javascript projects. BugsJS [19] selected 453 bug candidates, but they have not mentioned the theory behind that number. We also decided to use the same standard sample size calculation with 95% and a confidence interval of 5%. The required sample size was 385 bugs, but we assessed 500 bugs over five projects because we assumed we needed to filter out some bugs.

Those selected projects have different pull requests like bug fixes, features, sub-features, chores, refactored and document code commits. The Figure 3.2 figure shows the methodology for selecting candidate pull requests.

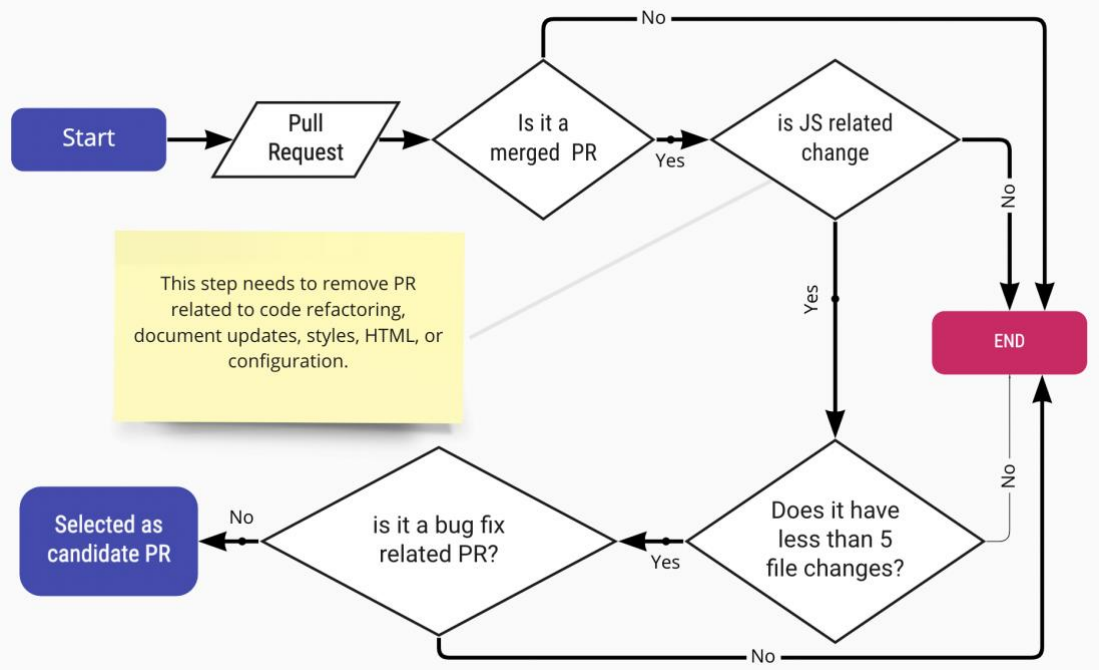


Figure 3.2 Bug Collection Process

Initially, we need to select only the merged pull requests because we need bug fixes that have been already reviewed and verified. Otherwise, there can be many open pull requests without merge, and we cannot guarantee whether those are valid or not. The BugsJS [19] also follows a similar approach. In the next steps, We removed code refactoring, document updates, styles, HTML, or configurations because our only intention is to collect only the bugs. Other studies have also followed that step to reduce the scope and pick only relevant commit or bugs [25, 17]

The BugsJS [19] says that they excluded the bug-fixing commits, which have more than three file changes and lines of 50, because they noticed those bugs were either complex or required domain knowledge to understand. However, we randomly picked a few bugs to understand that issue, but in our case, we decided to select bugs with less

than or equal to five file changes because sometimes bug fixes need to fix unit test codes and document files. So, it helped us to find more eligible bugs.

The next step is the tricky part in selecting candidate bugs because we need to identify only the bugs. N. Pingclasai et al. [27] reported that bug reports often come with incomplete information or poorly written reports. Also, Kim H et al. [36] report that 33.8% of the bug reports were misclassified. They say 39% of files marked as defects never had a bug. They suggest that manual inspections of the buggy code will be helpful to identify the bug PR, but it is biased to the reviewer. So, we decided that check the pull request code base, labels, titles, and descriptions to identify whether the pull request is a bug or not. Since we need to properly understand the code base or buggy code to annotate to Typescript, the manual inspection was helpful for the annotation task.

3.4 Typescript annotation

As we mentioned, we have already set up the Typescript configuration for each project. We need to check whether Typescript can detect the bug after annotating. Before we annotate, Figure 3.3 shows steps to follow to reduce the workload.

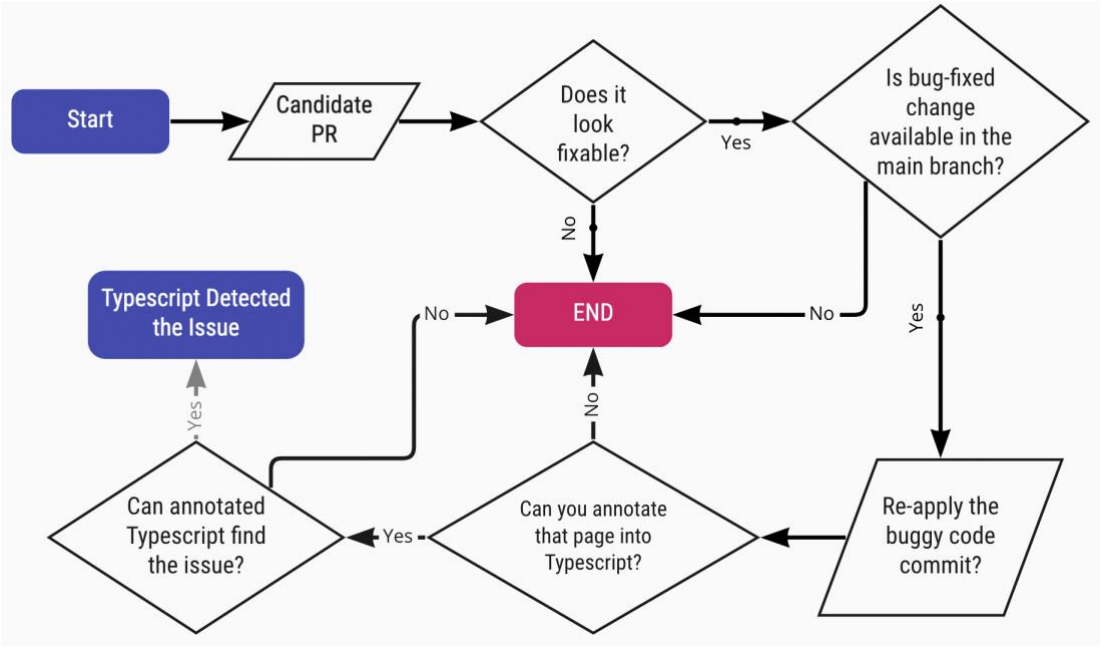


Figure 3.3 Code Conversion and validation methodology

Since we already filtered most of the issues such as feature implementations, documenting, and configurations, we can see only edge cases, logical errors, business logic, or security issues in this stage. If it looks impossible to fix by Typescript, we have to discard it.

We discarded bug fixes that are unavailable in the main branch because we assumed those changes are no longer valid, and it is not an easy task to check links to the reason unavailability of the code. We can see a different approach that was followed by P. Gyimesi et al. [19] to discard commits linked to more than one bug-fixing commit. Their concern they fixed in multiple steps or the initial try for fixing them was unsuccessful. In other words, commits can be no longer available or changed.

The next step is to apply the buggy code and annotate using Typescript. We had to convert that specific file to a typescript file and add relevant Type definitions to the bug. Then the code editor showed bugs if Typescript could identify. Applying a buggy version and analyzing is followed by the BugsJS [19]. The Z. Gao et al. [6] also annotated the Typescript to the buggy code base and checked whether it could determine the bug. The Figure 3.4 shows how we applied the buggy code version to the code base. The Figure 3.5 applied Typescript definition for the buggy code base.

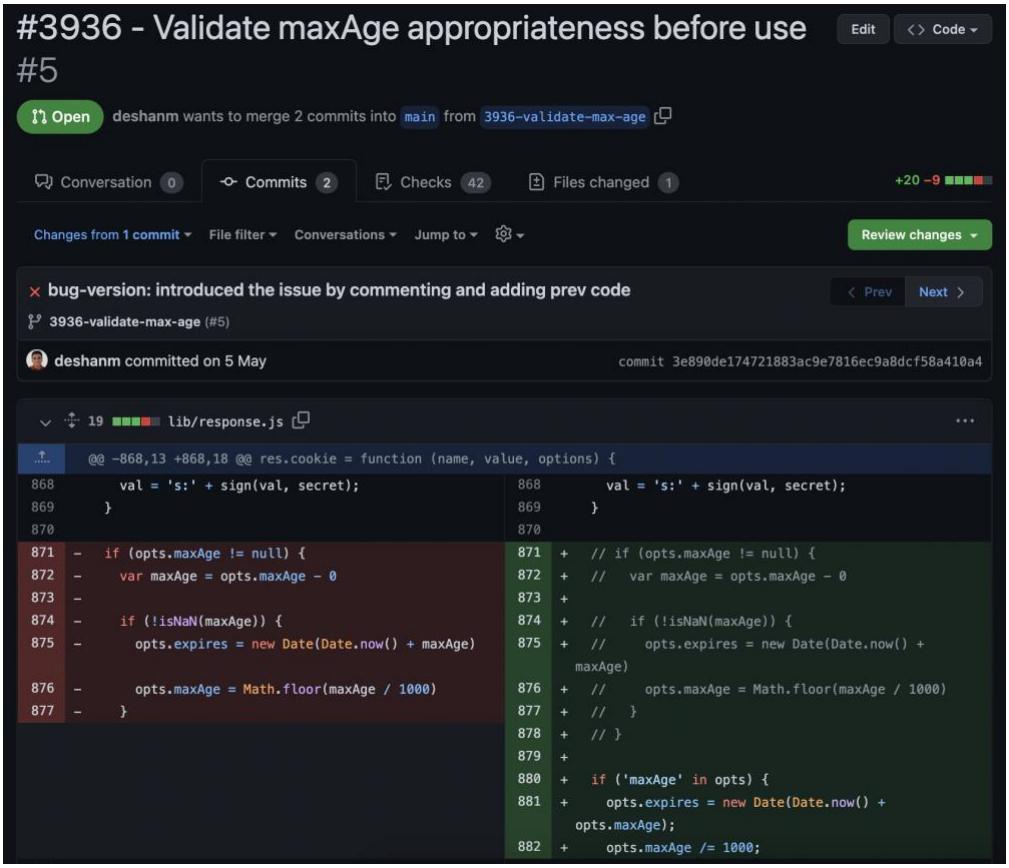


Figure 3.4 After applying buggy js change

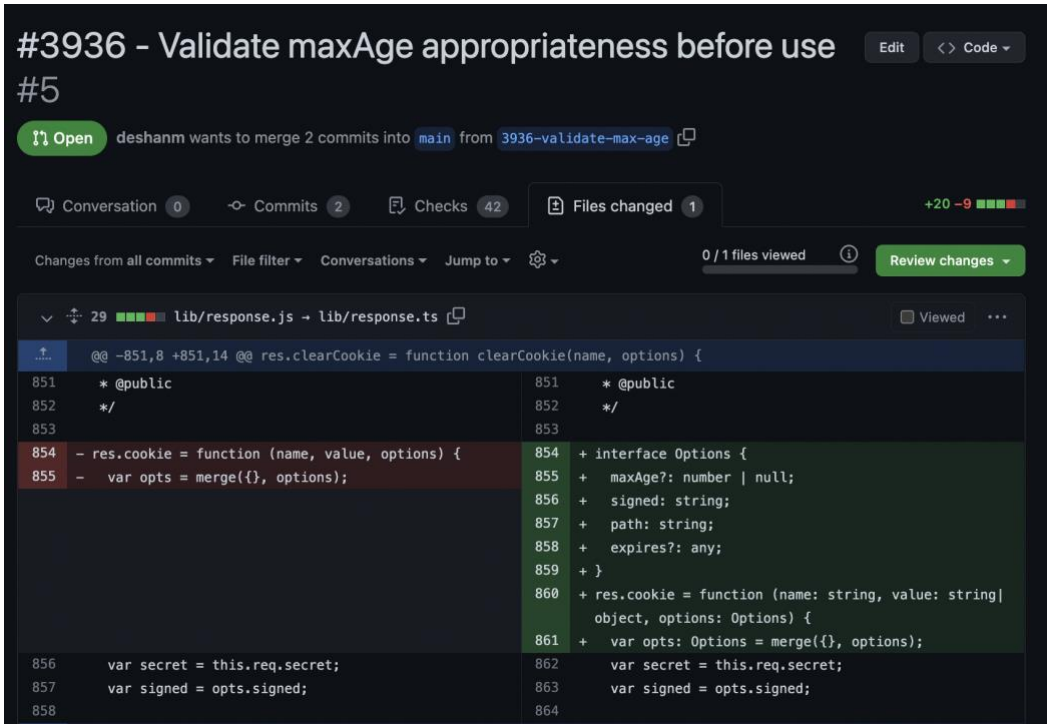


Figure 3.5 After applying typescript code annotation

The Figure 3.6 shows how we created a pull request for each Typescript fixable candidate bugs and added note with result screenshot for reference.

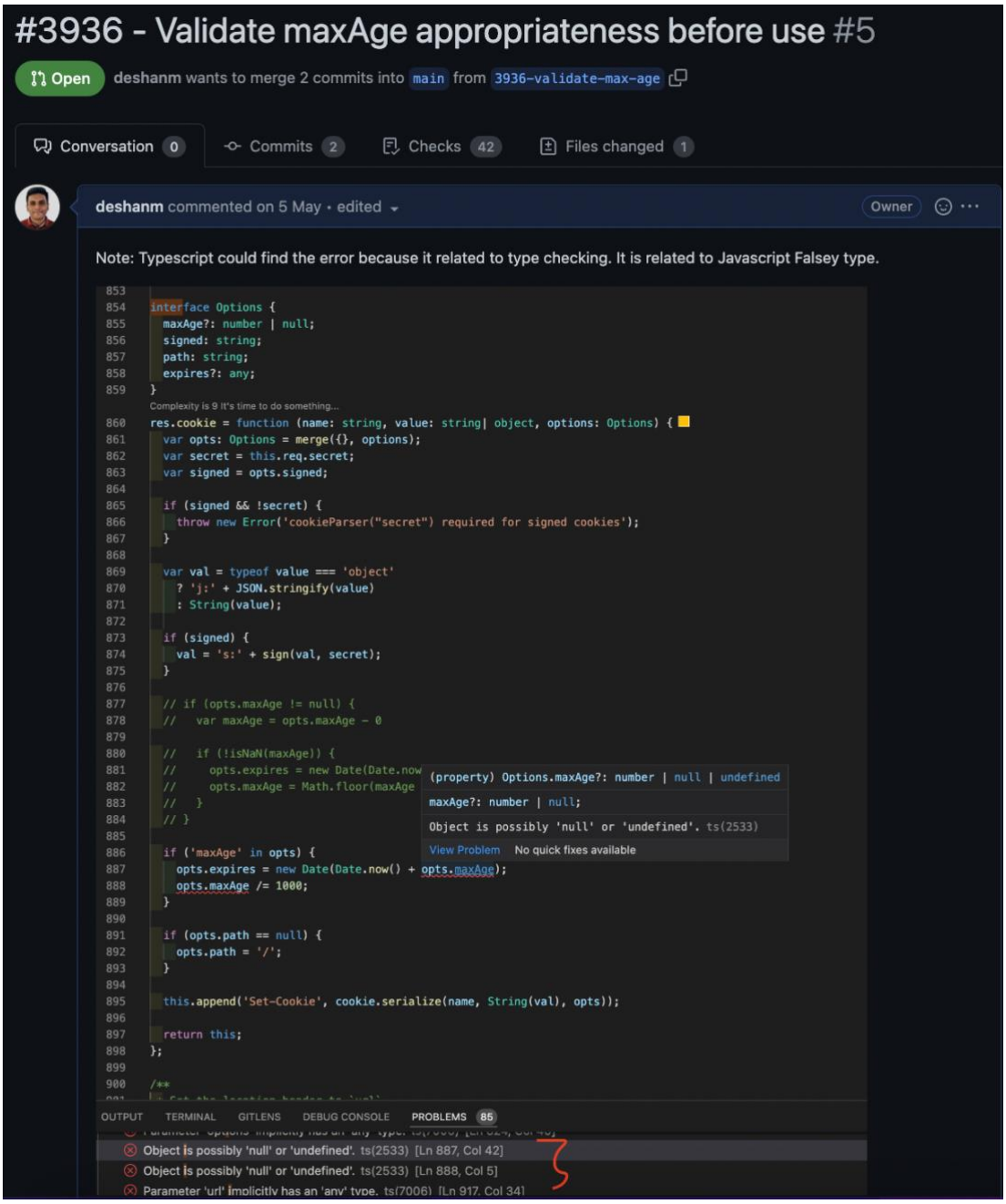


Figure 3.6 Github Sample Pull Request

3.5 Taxonomize & analyzing

Many studies use different methodologies to show bug classifications, such as taxonomy using faceted classification [19], Hanam et al.'s taxonomy [17], open coding [6], and F. Ocariza et al.'s [29] HOLOCRON. We decided to follow the open coding bug classification technique because a similar typescript research [6] paper followed it, and it is a simple classification compared to other reviewed classifications. However, in this study, the classification did not impact the output because most detected defects are Type Errors. After all, Typescript was designed to detect Type Errors. We used the GitHub labeling feature to label each pull request with those categories. We were able to categorized into 6 categories as Conditional Errors, Heavy Depend on Utils, Not A Bug, Unused, Global Variable Error and Type Error.

1. Conditional Errors: Most of the time, typescript cannot identify this type of issue because of conditional or logical errors. For example, depending on the option parameter, we need to get a predefined or resolved path from utils. The typescript cannot identify those issues.

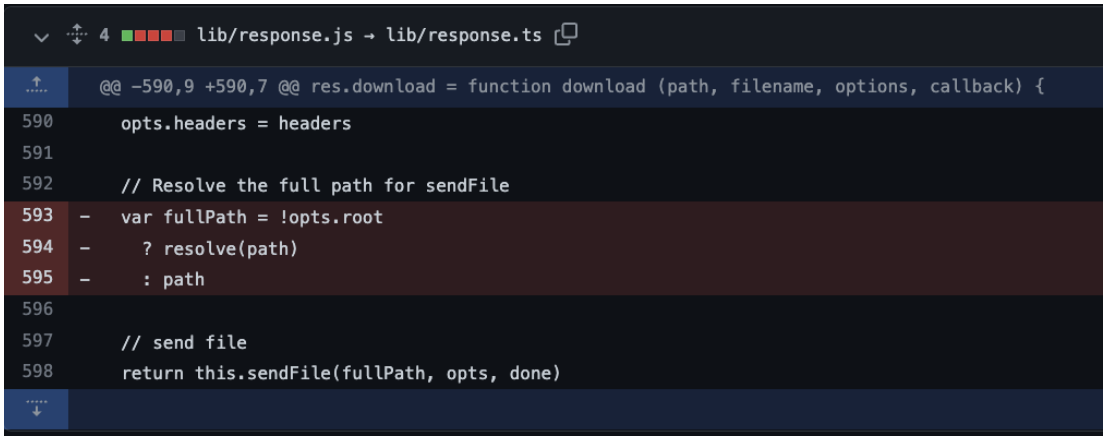


Figure 3.7 Conditional error

2. Heavy Depend on Utils: These issues heavily depend on the utility functions. Sometimes, the typescript can find if that util has type definitions. Most of the time, those bugs are complex to annotate. As the Figure 3.8 shows, the condition depends on the util function. If the code util has a type definition, typescript can identify it. If it does not have it, typescript cannot identify it.

```
if (isTypedArray(value)) {
  return result
}
```

Figure 3.8 Heavy depend on utils

3. Not A Bug: Although we applied all the filters before annotation, sometimes we get not a bug pull requests. The Figure 3.9 shows that use === instead of == because of the performance improvement. Usually, we use === instead of == because we need to check both value and type. However, in this case, they have changed it because of the performance.

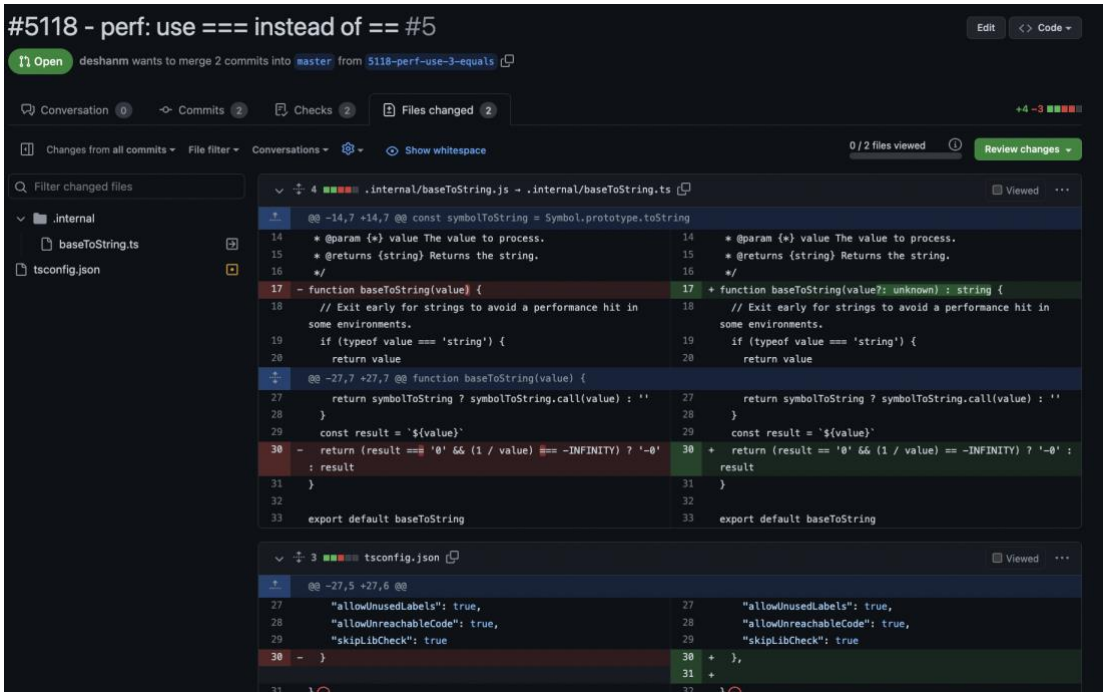


Figure 3.9 Not a bug

4. Unused: There can be code that we need to clean. For example, we might need to remove the function's unused optional parameter.

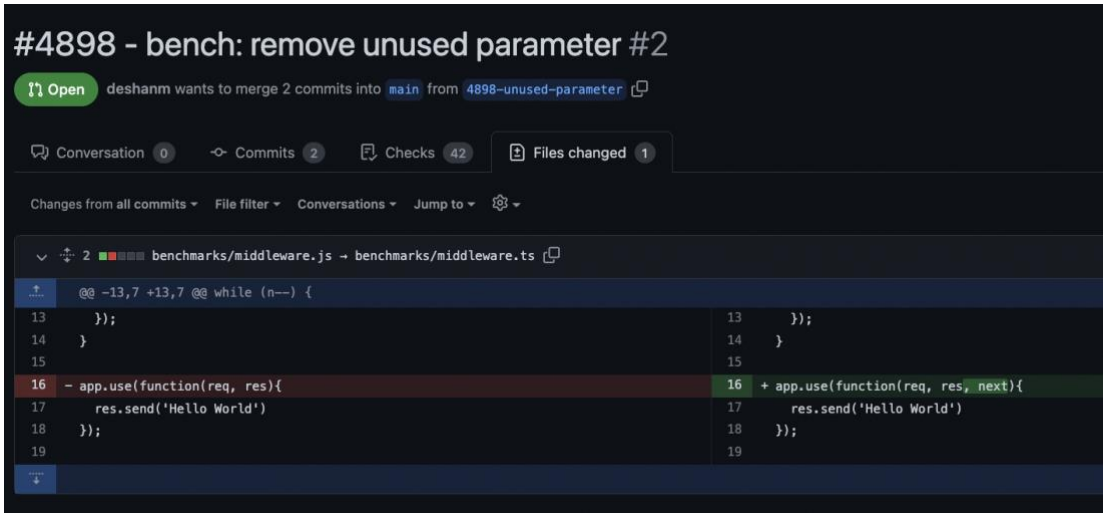


Figure 3.10 Unused

5. Global Variable Error: Global polluted variable makes the javascript complex—for example, global value changes or window changes. Most of the time, typescript cannot identify, or we need to spend more time changing the global variables.

Figure 3.11 Global variable

6. Type Error: Most of the time, typescript can identify this kind of issue because the primary responsibility of the type system is to handle the type errors. The Figure 3.12 show how it identifies the type errors.

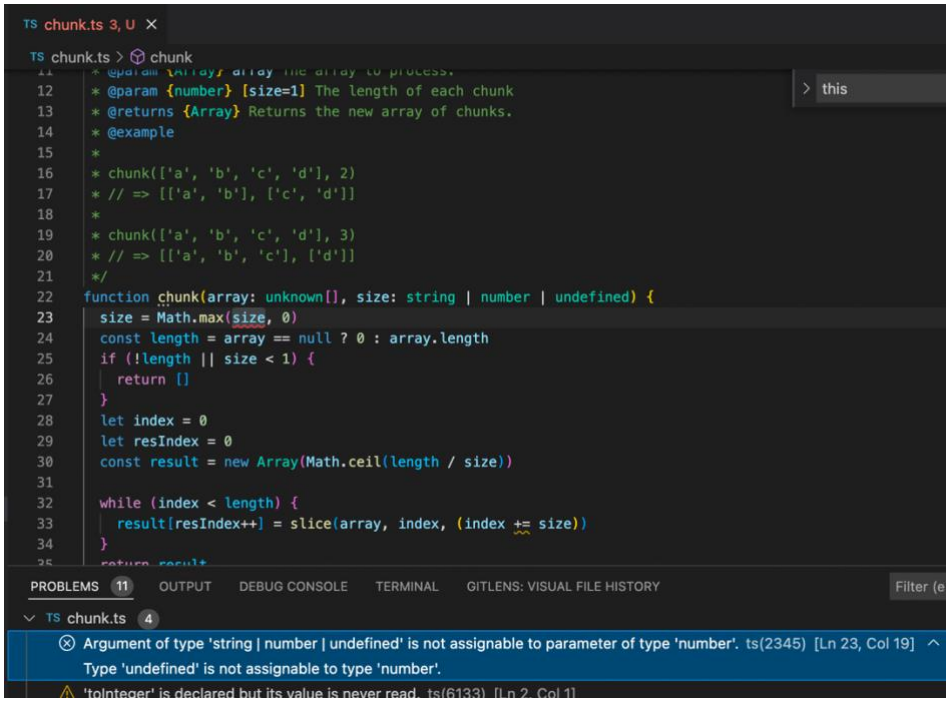


Figure 3.12 Type error

Other than the above issues, we added another two labels to identify the Typescript success or not (ts-wins and ts-fails) only for the reference.

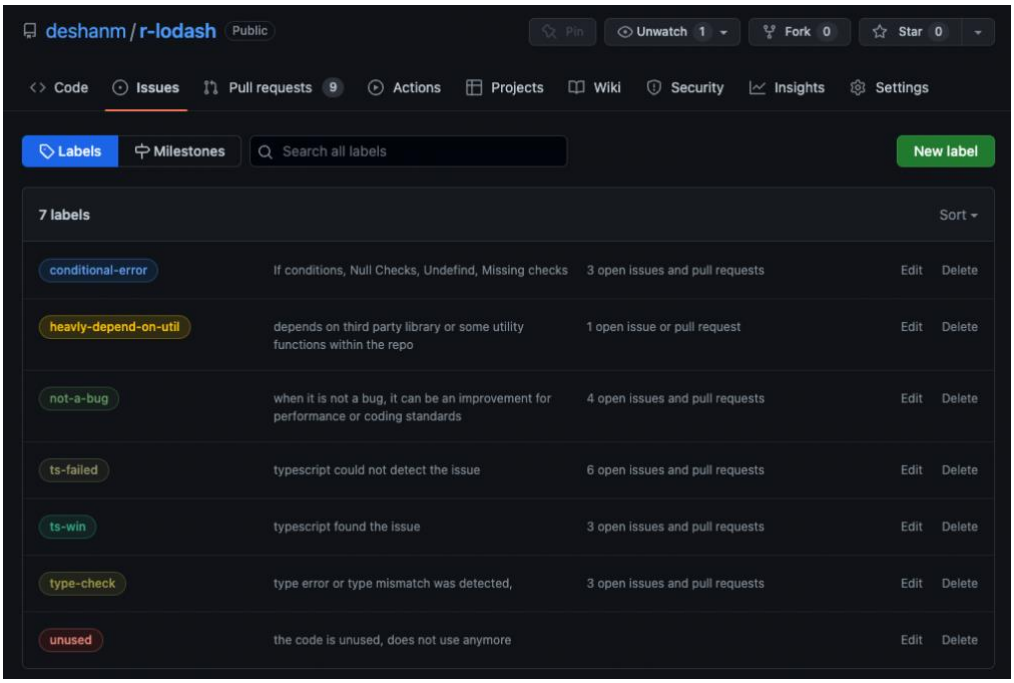
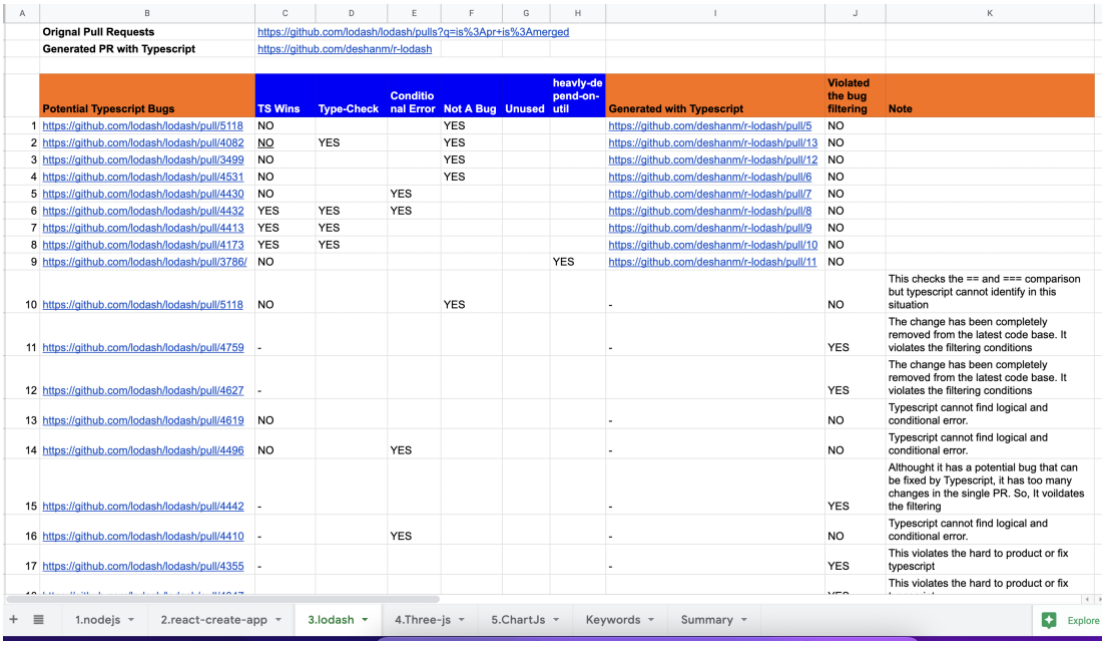


Figure 3.13 Github labels

We created a sheet for each project, including potential typescript fixable-bug pull requests, issue labels, generated typescript annotation, and filter violations and notes. Even after the initial bug candidate filter, we had to discard some of the bugs in the annotation state. For those bugs we created a column called “Violated the Bug Filtering”.



	Orignal Pull Requests	https://github.com/lodash/lodash/pulls?q=is%3Apr+is%3Amerged								
	Generated PR with Typescript	https://github.com/deshanm/r-lodash								
	Potential Typescript Bugs	**TS Wins**	**Type-Check**	**Conditional Error**	**Not A Bug**	**Unused**	**heavly-depend-on-util**	**Generated with Typescript**	**Violated the bug filtering**	**Note**
1	https://github.com/lodash/lodash/pull/5118	NO			YES			https://github.com/deshanm/r-lodash/pull/5	NO	
2	https://github.com/lodash/lodash/pull/4082	NO	YES		YES			https://github.com/deshanm/r-lodash/pull/13	NO	
3	https://github.com/lodash/lodash/pull/3499	NO			YES			https://github.com/deshanm/r-lodash/pull/12	NO	
4	https://github.com/lodash/lodash/pull/4531	NO			YES			https://github.com/deshanm/r-lodash/pull/6	NO	
5	https://github.com/lodash/lodash/pull/4430	NO		YES				https://github.com/deshanm/r-lodash/pull/7	NO	
6	https://github.com/lodash/lodash/pull/4432	YES	YES	YES				https://github.com/deshanm/r-lodash/pull/8	NO	
7	https://github.com/lodash/lodash/pull/4413	YES	YES					https://github.com/deshanm/r-lodash/pull/9	NO	
8	https://github.com/lodash/lodash/pull/4173	YES	YES					https://github.com/deshanm/r-lodash/pull/10	NO	
9	https://github.com/lodash/lodash/pull/3786/	NO					YES	https://github.com/deshanm/r-lodash/pull/11	NO	
10	https://github.com/lodash/lodash/pull/5118	NO			YES			-	NO	This checks the == and === comparison but typescript cannot identify in this situation
11	https://github.com/lodash/lodash/pull/4759	-						-	YES	The change has been completely removed from the latest code base. It violates the filtering conditions
12	https://github.com/lodash/lodash/pull/4627	-							YES	The change has been completely removed from the latest code base. It violates the filtering conditions
13	https://github.com/lodash/lodash/pull/4619	NO						-	NO	Typescript cannot find logical and conditional error.
14	https://github.com/lodash/lodash/pull/4496	NO		YES				-	NO	Typescript cannot find logical and conditional error.
15	https://github.com/lodash/lodash/pull/4442	-						-	YES	Althought it has a potential bug that can be fixed by Typescript, it has too many changes in the single PR. So, It voildates the filtering
16	https://github.com/lodash/lodash/pull/4410	-		YES				-	NO	Typescript cannot find logical and conditional error.
17	https://github.com/lodash/lodash/pull/4355	-						-	YES	This violates the hard to product or fix typescript
										This violates the hard to product or fix

1.nodejs | 2.react-create-app | 3.lodash | 4.Three-js | 5.ChartJs | Keywords | Summary | Explore

Figure 3.14 Final Results Sheet

4. RESULTS & DISCUSSION

4.1 Challenges in identifying gugs

One of the most critical challenges was to find set of potential bugs that Typescript can detect. We assessed 500 bugs from those selected repositories with our set of pre-defined rules set.

The Figure 4.1 chart shows how it has been distributed just merged vs. total (open + closed) pull requests. As it shows, compared to total pull requests, merged pull requests are too low. All the repositories had a minimum of 500 merged pull requests, but most were discarded during the bug candidate selection filtering. Since we needed only the merged bug pull requests, we had to discard feature implementations, document updates, configuration updates, and none Javascript updates.

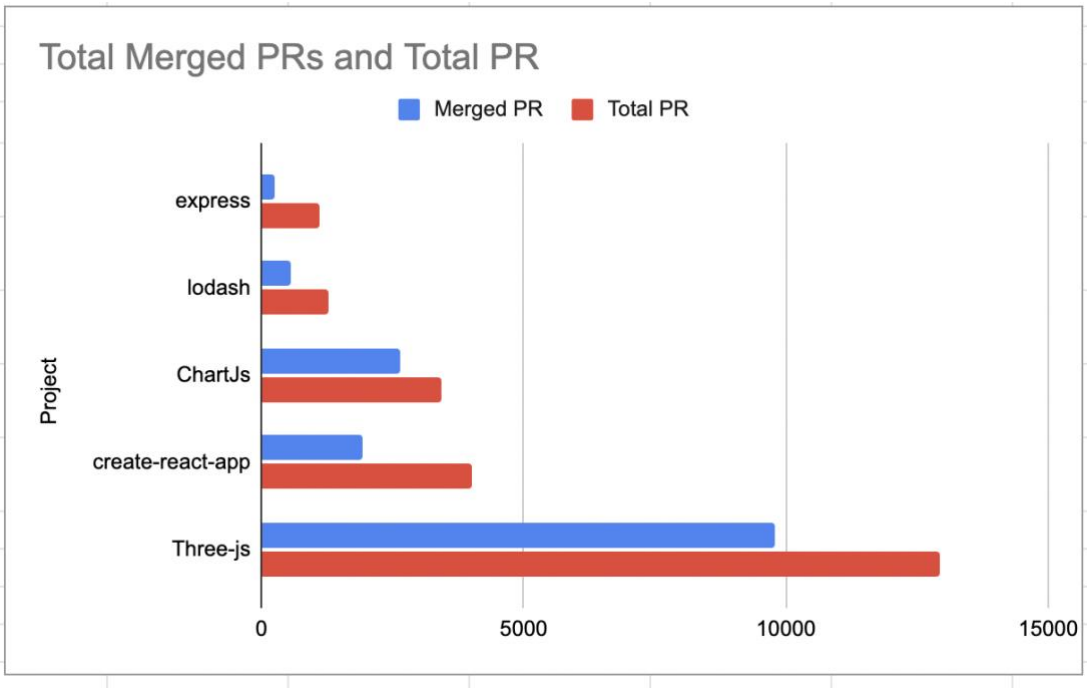


Figure 4.1 Merged VS Total PR

Since most of the pull requests are related to the documents, new features and enhancements, we get very few numbers of bug pull requests. As example, if we filter the react-react-app repository to filter only bug labels (react-create-app filtered pull requests), we got only 273 out of 1934 total pull requests. That means 14.11%. In that also includes a lot of unnecessary pull requests like web pack configurations, script

related configs, package updates, CSS updates, HTML updates, naming convention changes or browser related issues. Apart from those filtering, we ignored older commits, pull requests with complex codes, pull requests with a lot of file changes. Finally, we could get only 75 bugs that can be fixed by the Typescript. If we compare with other research papers, “To Type or Not to Type: Quantifying Detectable Bugs in JavaScript” had only 80 bugs [6]. The “BUGSJS: A Benchmark and Taxonomy of Javascript Bugs” [19] had 400 bugs but they studied all the bugs regardless of Typescript. Also, they collected those 400 bugs after studding 398 projects.

4.2 Typescript configuration impacts on results

The results of those selected projects provide different results because it is difficult to compare one factor with completely different factors such as the experience of the developers, coding standards, automation, contributors' commitment, community contributions, usage, and project start time. From the Bogner el al.'s [26] lesson, we realized that Typescript configuration plays a significant role in detecting bugs. Adding a strict Typescript configuration will be able to detect more issues [26], but the developers have to spend more time defining Types.

For example, the Figure 4.2 code shows find a user by his name. Then log the name of the found user. However, the loggedInUser can be null and it cannot get the age from the null value. This configuration cannot find that kind of issue because we have changed the “strictNullCheck” to false.

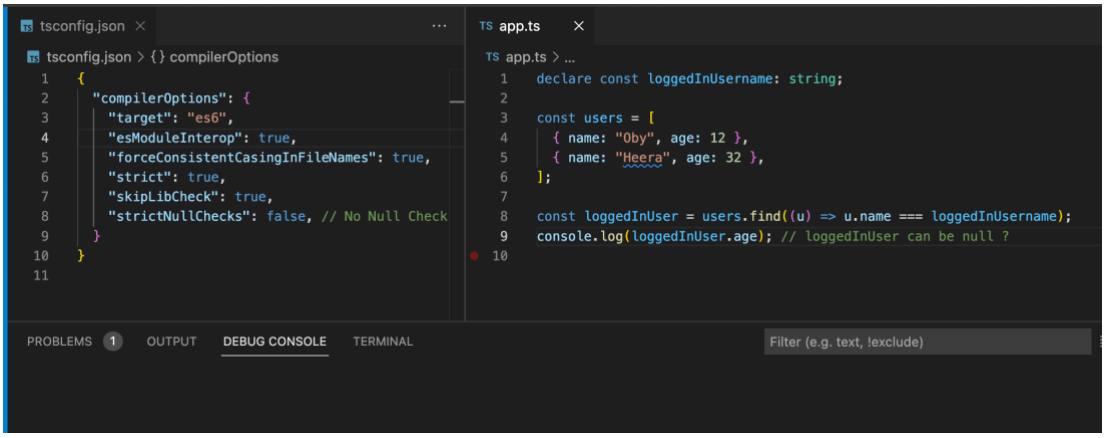


Figure 4.2 Code without strick nullcChecks

If we change the “strictNullCheck” to true, we can see the error at the compile or IDE as the Figure 4.3 screenshot.

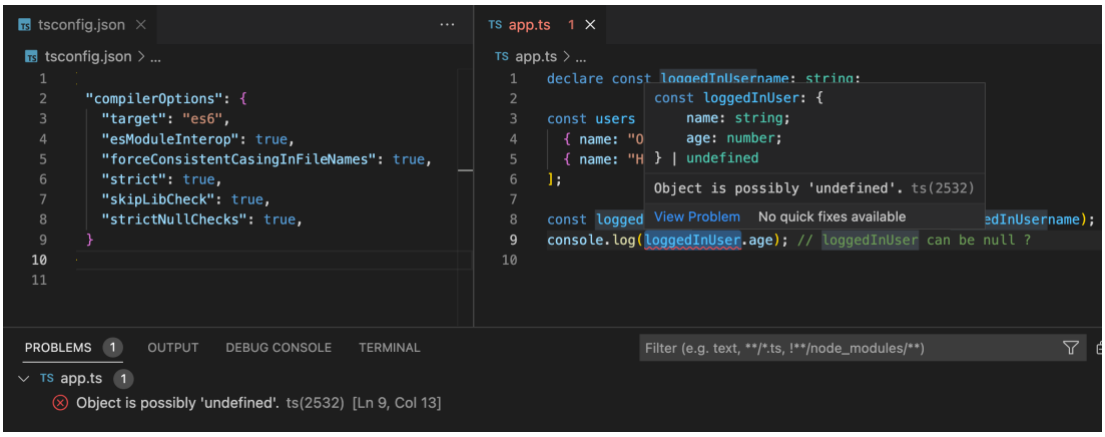


Figure 4.3 Code with strict null checks

We used Figure 4.4 Typescript configuration for all the projects which helped us to produce a fair result. The Bogner et al. [26] also followed the same methodology in their research.

Figure 4.4 Default tsconfig.json

4.3 Results of selected pull requests

4.3.1 ChartJS

The ChartJs had a total of 2,643 merged pull requests, we analyzed 100 bugs after applying all the filtering, and we found 20 bugs that can be fixed by Typescript. The Figure 4.5 shows the results classification after analyzing and annotating the code into Typescript for each bug.

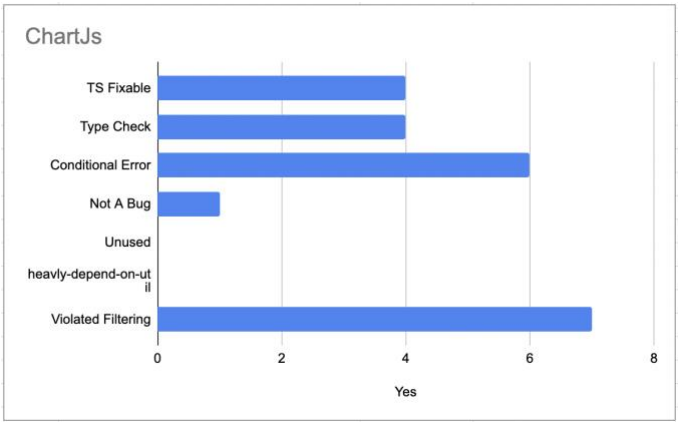


Figure 4.5 Chart JS

4.3.2 ThreeJS

Among the selected Javascript libraries, the ThreeJS have the highest merged pull requests (9,781). However, most of them are related to HTML, CSS, Documents, and Test changes compared to Javascript. We analyzed 200 bugs using after applying all the filtering in the GitHub then we found 15 bugs that have the potential to be fixable by Typescript.

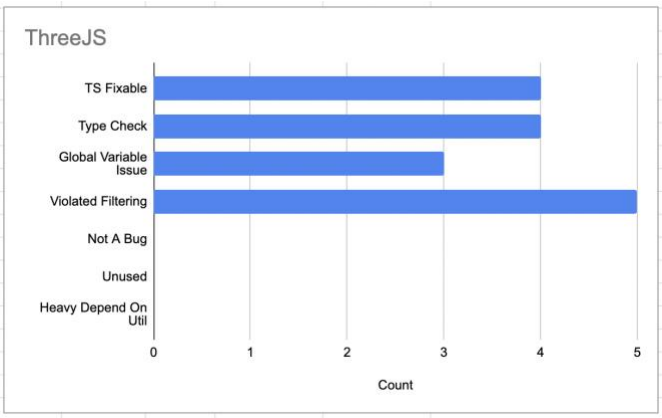


Figure 4.6 ThreeJS

4.3.3 Create react app

We analyzed 120 bugs after applying all the filtering in Github we found only five bugs that can be fixed by Typescript. This repo is very different from other repositories because the number of errors was small. The paid full-time Facebook developers maintain this repo. Usually, developers in those big tech companies are highly talented and experts in their field. Since it is a full-time job, they can focus more on projects. Probably that is why the error count was minimal. Also, this repo is a generator. Since it is a generator, users do not involve it much because once it generates the code base, developers use and customize it. However, after analyzing and converting the code into Typescript for each bug, the Figure 4.7 shows the output.

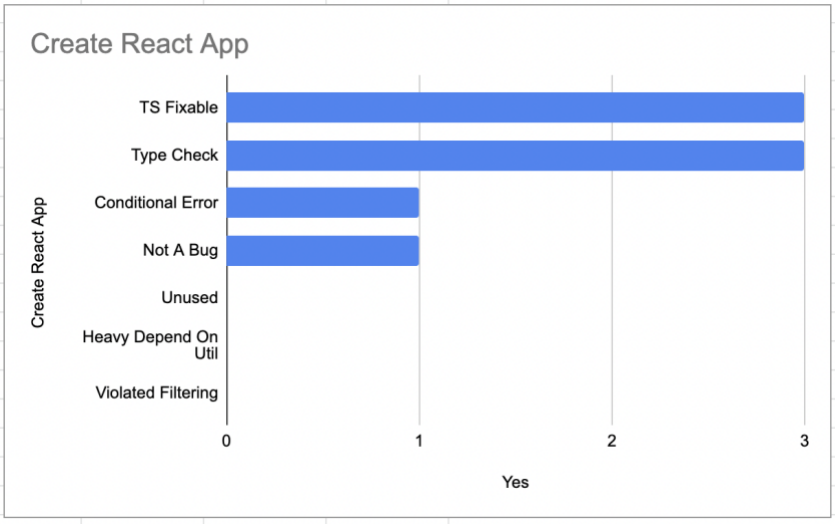


Figure 4.7 Create React App

4.3.4 Express

The Expresses is a popular, simple, and stable framework with only 256 merged pull requests. We analyzed 80 bugs after applying all the filtering in GitHub, and we found 15 bugs that have the potential to be fixable by Typescript. The Figure 4.8 screenshot shows the results.

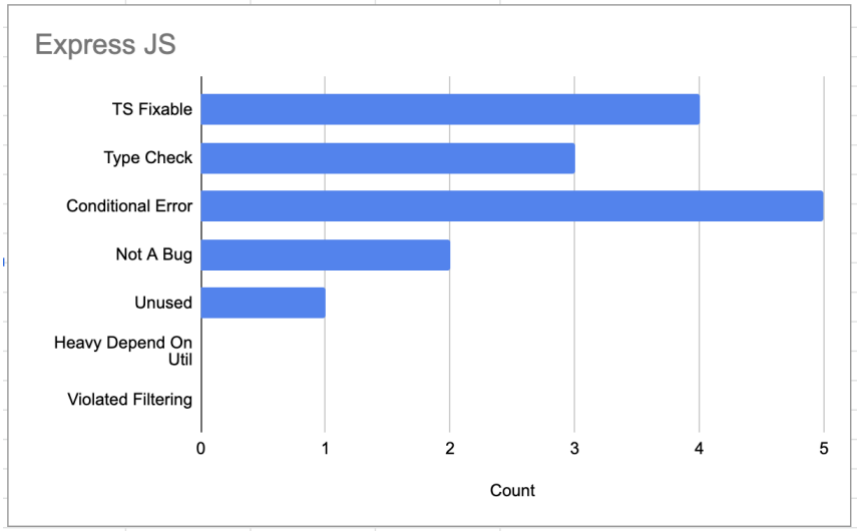


Figure 4.8 Express JS

4.3.2 Lodash

The Lodash is a widely used utility library in the Javascript community. We analyzed 80 bugs after applying all the filtering in GitHub, and we found 20 bugs that have the potential to be fixable by Typescript. We found five merged pull requests that are not a bug. It is the highest number we got compared to other projects.

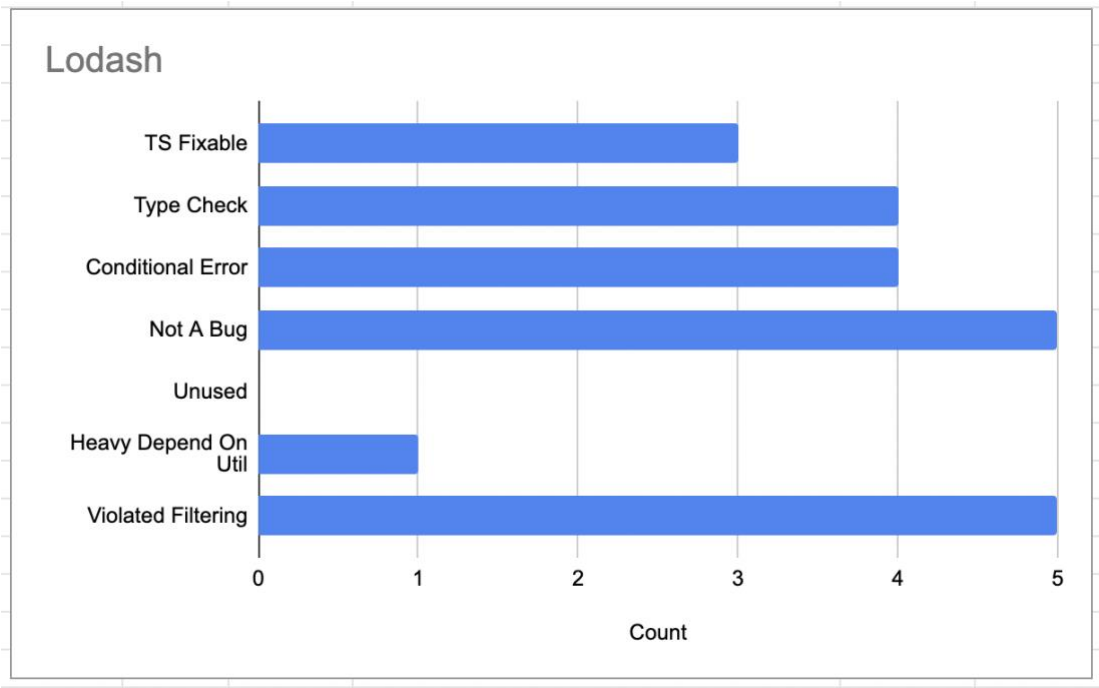


Figure 4.9 Lodash

4.5 Results and discussion

The Table 2.1 shows the summary of the initially reviewed pull requests and analyzed potential Typescript fixable bugs. Those initially reviewed pull requests are not just reviewed, all of them were reviewed after applying GitHub filters (ex: is:merged label: ”bug”, ”fix”, ”issue” -label: “doc”, “docs”) to avoids unnecessary pull requests.

Table 4.1 Reviewed vs potential TS fixable PR

Project	Reviewed PRs	Final Candidate Bugs	%
Express	80	15	18.75
React Create App	100	5	5
Lodash	100	20	20
ThreeJS	120	15	12.5
ChartJs	100	20	20
	500	**75**	**15**

With the help of those filters, were could identify 75 potential Typescript fixable pull requests. Although those pull requests have the potential to be fixable by Typescript, we had to annotate all of the bugs with Typescript types to check whether it works with Typescript.

However, when we consider the BugsJS [19] selected projects, they had five projects under ten bugs, and ESlint has 333 bugs. That means 73.5% of bugs are Eslint bugs which are biased to that project. In our research, the react-create-app has the lowest bugs count (5%), and Lodash has the highest (20%), which is properly distributed among the projects and helps to get unbiased results.

While we were annotating, we found one interesting issue in the Typescript. The Figure 4.10 screenshot is taken from the Typescript GitHub Repository [37]. It says that JSON.stringify method always returns string values.

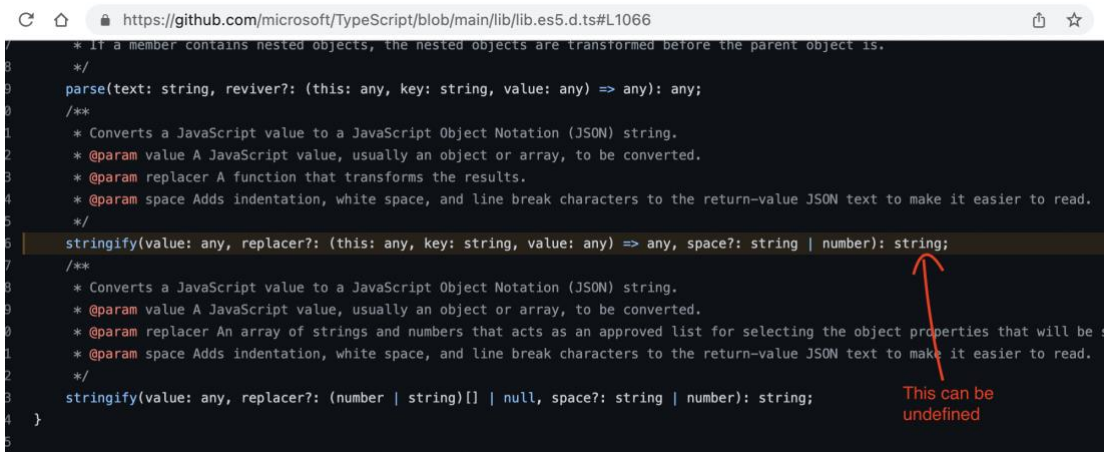


Figure 4.10 Typescript Bug – Stringify. Adapted from [37]

Then we did an experiment with the online Javascript tool, we got the Figure 4.11 Typescript Bug Reproduce – Stringify results. If you send undefined, it will return undefined. Due to this Typescript definition error in the Typescript repo, one of the selected pull requests failed to identify by the Typescript.

Figure 4.11 Typescript Bug Reproduce – Stringify

Besides the Typescript bug we found, we did not face any issues from the Typescript side for those selected pull requests.

The Table 4.2 Comparison TableTable 4.2 and Figure 4.12 show how those selected pull requests distributed with our categories. As it shows, most of the selected pull requests have violated the health condition of the pull request to review by Typescript. That means those pull requests are not compatible with the current main branch because those code base has been completely removed from the repo. We got only 5

bugs 3 categories which for unused, which heavily depends on utils and global variable issues. Most of the categories are properly distributed except those three categories. When we compare this with Z. Goa [6] categories, 75% of bugs are in SpecError category and other 15% bugs are distributed within 9 categories. Also, we noticed 12 bugs that were not actually bugs, there were code refactoring or code improvements.

Table 4.2 Comparison Table

Title	Express	React Create App	Lodash	Three JS	Chart JS	Total
TS Fixable	3	3	3	4	4	17
Type Check	4	3	4	4	4	19
Conditional Error	4	1	4		6	15
Not A Bug	5	1	5		1	12
Unused	1				0	1
Heavy Depend On Util			1		0	1
Violated Filtering	5		5	5	7	22
Global Variable Issue				3		3
Total Issues	15	5	20	15	20	75

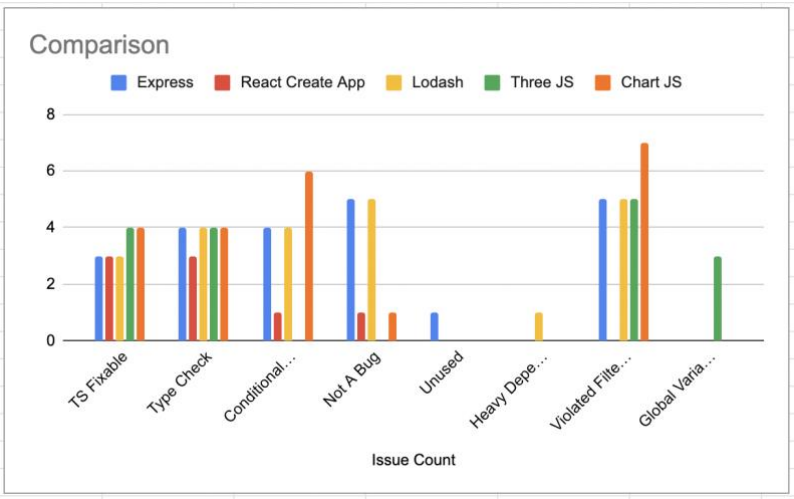


Figure 4.12 Comparison Chart

Our main object was to identify “How impactful is it to use Typescript over Javascript to detect defects?”. In a summary, we could found 22.7% of defect by using Typescript over Javascript. It is huge compared to similar research done by Z. Gao, C. Bird, and E. T. Barr [6] which showed 15% detectable ability using Typescript. However, their research was done by using Typescript 2.0 version but we used Typescript 4.0. In their research, they started the experiment with Typescript 1.0 and then moved to Typescript

2.0 because it could detect more bugs. Since we used the latest version, we probably got improvement. Also, those numbers are depending on the Typescript configuration.

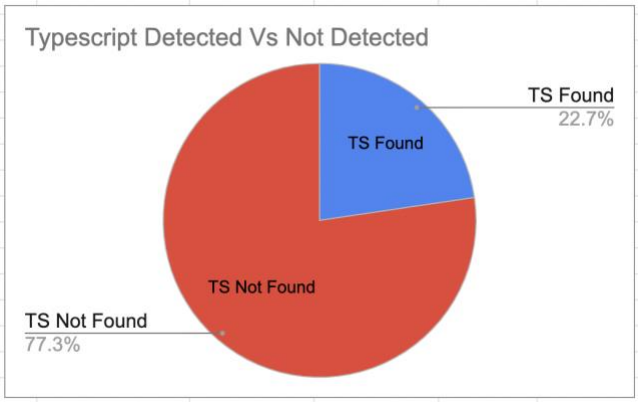


Figure 4.13 Typescript found vs not found bugs

Airbnb said that "38% of bugs in the Airbnb codebase were preventable with TypeScript" [11]. It is questionable in their statement because they have not publicly provided relevant evidence for that. However, their approach is entirely different from all the other research papers we reviewed because they migrated entire Javascript projects into Typescript. We assumed the result might be impacted because of their domain knowledge about the project, and they have paid full-time engineers to maintain it. We picked projects that multiple organizations or individual contributors developed in our research. So, it is hard to compare with the Airbnb experience. However, the answer for our research objective we assume that 22.7% is something middle between Aibnb and Z. Gao [6] et al.

We believe this study will be great evidence to consider using Typescript over Javascript in the future to reduce the notable number of bugs.

5. SUMMARY AND CONCLUSION

Javascript is a dynamically typed programming language that gives freedom to ignore types and build applications quickly. Still, Javascript is a poor language for identifying bugs at the compile time and maintaining large applications because of the dynamic behavior. The developers can use Typescript to add type syntaxes on top of JavaScript as a solution. However, there are not enough studies showing the impact that Typescript can make on detecting bugs. There are only a few studies related to Typescript because it is a relatively new language. After all, it was introduced in 2012. This was challenging research because there were not enough studies to compare or gain knowledge. Since there is no proper tool to convert Javascript to Typescript, we had to annotate each bug-fix code manually, which is a time-consuming task. It limited us to select only five projects, but other empirical studies also had limited projects.

However, some studies had sampled many projects, but results are bias to one or two projects. For example, they picked 73.5% from one project and other bugs from the other nine projects. The distribution is biased to a specific project. In our research, the selected bugs from each project were adequately distributed that ranging between 5-20%. However, our project and bug selection criteria are similar to most projects we reviewed.

Although we needed only 385, We initially reviewed 500 pull requests over five projects. After applying all the criteria to the bugs, we found 75 candidate bugs we assumed Typescript could fix. The results are encouraging; using Typescript over Javascript could have detected 22.7% of bugs. As far as we know, this is one of the few studies related Typescript to identifying bug impact. Also, compared to other available studies, the result is close to them. We believe this study will be great evidence to consider using Typescript over Javascript in the future to reduce the notable number of bugs.

Since we selected only five projects due to the limitations, it is not a generalized result. Different skillful engineers develop a Javascript application with different configurations and tools. Also, our selected projects are either libraries or tools. Most

developers use those tools to build a project, and they don't build libraries or tools. However, we followed a standard empirical style to provide unbiased results.

Javascript is a widely used language with many implementations, standards, and tools. Even without Typescript, some tools like ESLint and JSHint can identify some defects. Those tools help to reduce bugs and enforce code quality. Also, unit testing and testing coverage help to reduce the bug rate of the project. In this research, we did not count those while analyzing the impact of Typescript.

Concerning the bugs, we manually classified and annotated them into Typescript. There are many different ways to Annotate, and there are advanced patterns in Typescript. Sometimes the domain knowledge is required to follow best practices to build high-quality Typescript projects. Since it is a manual process, each annotated bug's results can slightly differ.

In the future, we plan to expand this to identify other factors in Typescript's impact on the code base and the developers. Adapting or using Typescript costs more than Javascript because developers must write additional code and maintain the complexity. Usually, more senior engineers prefer to use Typescript because of that.

Since we have to add extra code in Typescript, it changes the understandability, readability, and number of lines. If it increases the code complexity, the developers must reduce it using various ways, such as splitting code into small codes or rewriting the code. Otherwise, it will be harder to onboard new team members, causes more rework, and produces more susceptibility to bugs. So, we plan to study how Typescript impacts code complexity and the number of code lines. We can easily bug smelling tools like SonarQube to analyze the code after converting Javascript code into Typescript.

Since the process of Annotating is time-consuming in this research, we plan to use an automatic Typescript conversion tool to automate this. We can quickly convert a large sample size if we can use an automated tool. Apart from that, it will help to convert all the projects similarly. Otherwise, manual annotation might be different from project to project.

References

[1] D. Tomassi, "Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs," *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineerin,* 2018.

[2] A. Rio and B. A. Fernando, "Detecting Sudden Variations in Web Apps Code Smells' Density: A Longitudinal Study," *Quality of Information and Communications Technology,* pp. 82-96, 2021.

[3] Fard, Amin M, Mesbah A., "JSNOSE: Detecting JavaScript Code Smells," *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM,* pp. 116-125, 2013.

[4] T. V. D. W. S. J. E. Roehm, "Evaluating Maintainability Prejudices with a Large-Scale Study of Open-Source Projects," *Springer International Publishing,* pp. 151--171, 2018.

[5] Elder V., Andrea D., Marcelo M., "A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where," *IEEE Transactions on Software Engineering,* vol. 47, no. 1, pp. 17-66, 2021.

[6] T. Amanatidis, A. Chatzigeorgiou, "Studying the Evolution of PHP Web Applications," *Butterworth-Heinemann,* vol. 72, no. 20, p. 48–67, 2016.

[7] Z. Gao, C. Bird and E. Barr, "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript," *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE),* 2017.

[8] D. Flanagan, JavaScript - The Definitive Guide, Sebastopol: O'Reilly Media, 2002.

[9] B. Pierce, Types and Programming Languages, Massachusetts: The MIT Press, 2002.

[10] Meijer, Erik, Drayton, "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages," *OOPSLA'04 Workshop on Revival of Dynamic Languages,* 2004.

[11] "How Codebase Health Impacts Hiring and Retention 2021 Report," Stepsize, 2022. [Online]. Available: https://www.stepsize.com/how-codebase-health-impacts-hiring-and-retention-2021-report. [Accessed 03 02 2022].

[12] B. Couriol, "Airbnb Releases Tool to Convert Large Codebases to Typescript," InfoQ, 2022. [Online]. Available: https://www.infoq.com/news/2020/08/airbnb-Typescript-migration/. [Accessed 03 02 2022].

[13] "TypeScript at Google," Youtube.com, 2018. [Online]. Available: https://www.youtube.com/watch?v=sjov1k5jexA. [Accessed 03 02 2022].

[14] "TypeScript at Slack - Slack Engineering," Slack Engineering, 2017. [Online]. Available: https://slack.engineering/Typescript-at-slack/. [Accessed 04 02 2022].

[15] "Stack Overflow Developer Survey 2021," Stack Overflow, 2022. [Online]. Available: https://insights.stackoverflow.com/survey/2021. [Accessed 2 02 2022].

[16] D. Spinellis, "Choosing a programming language," *IEEE Software,* vol. 23, no. 4, pp. 62-63, 2006.

[17] "flow-bin | npm trends," Npmtrends.com, 2022. [Online]. Available: https://www.npmtrends.com/flow-bin-vs-Typescript. [Accessed 2 02 2022].

[18] "New Node.js Foundation Survey Reports New "Full Stack" In Demand Among Enterprise Developers | Node.js," Node.js, 2022. [Online]. Available: https://nodejs.org/uk/blog/announcements/nodejs-foundation-survey/. [Accessed 04 02 2022].

[19] Q. Hanam, F. Brito and A. Mesbah, "Discovering bug patterns in JavaScript," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* no. 156, p. 144, 2016.

[20] G. Catolino, F. Palomba, A. Zaidman and F. Ferrucci,, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software,* no. 181, p. 165, 2019.

[21] P. Gyimesi et al., "BUGSJS: a benchmark and taxonomy of JavaScript bugs," *Software Testing, Verification and Reliability,* vol. 31, 2021.

[22] Dalton J., Patricia M., Wilkerson A., "Investigating Test Smells in JavaScript Test Code," *Brazilian Symposium on Systematic and Automated Software Testing,* vol. 2, no. 10, p. 36–45, 2021.

[23] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering,* vol. 37, no. 5, pp. 649-678, 2011.

[24] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis and N. Malevris,, "Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Stud," *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM),* 2006.

[25] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering,* vol. 25, no. 4, pp. 557-572, 1999.

[26] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," *2013 35th International Conference on Software Engineering (ICSE),* pp. 752-761, 2013.

[27] "Find Bugs in Java Programs," findbugs.sourceforge.net, 02 06 2015. [Online]. Available: http://findbugs.sourceforge.net/. [Accessed 20 03 2022].

[28] Bogner, Justus & Merkel, Manuel, "To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub," *022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR),*, pp. 658-669, 2022.

[29] N. Pingclasai, H. Hata and K. Matsumoto, "Classifying Bug Reports to Bugs and Other Requests Using Topic Modeling," *2013 20th Asia-Pacific Software Engineering Conference (APSEC),* vol. 2, pp. 13-18, 2013.

[30] F. Ocariza, K. Pattabiraman and A. Mesbah, "Detecting unknown inconsistencies in web applications," *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[31] C. Le Goues, M. Dewey-Vogt, S. Forrest and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each"," *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[32] J. Wang et al., "A comprehensive study on real world concurrency bugs in Node.js," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[33] T. Leesatapornwongsa, J. Lukman, S. Lu and H. Gunawi, "TaxDC," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[34] M. Kumari, A. Misra, S. Misra, L. Fernandez Sanz, R. Damasevicius and V. Singh, "Quantitative Quality Evaluation of Software Products by Considering Summary and Comments Entropy of a Reported Bug," *Entropy*, vol. 21, no. 1, p. 91, 2019.

[35] "Code Quality and Code Security," SonarSource SA, 2022. [Online]. Available: https://www.sonarqube.org/. [Accessed 02 04 2022].

[36] G. Pinto, F. Casto and W. Torres, "A study on the most popular questions about concurrent programming | Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools," in *ACM Conferences*, 2022.

[37] K. Herzig, S. Just and A. Zeller, "Herzig, Kim and Just, Sascha and Zeller, Andreas," *2013 35th International Conference on Software Engineering (ICSE)*, pp. 392-401, 2013.

[38] D. D., "lib.es5.d.ts," microsoft, [Online]. Available: https://github.com/microsoft/TypeScript/blob/main/lib/lib.es5.d.ts#L1066. [Accessed 02 02 2022].

[39] A. Bachmann, C. Bird, F. Rahman, P. Devanbu and A. Bernstein,, "The missing links," *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE*, 2010.

[40] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," *Association for Computing Machinery*, vol. 42, no. 2, p. 329–339, 2008.

[41] S. Hanenberg,, "An experiment about static and dynamic type systems," *ACM SIGPLAN Notice*, vol. 45, no. 10, pp. 22-35, 2010.

[42] "GitHub - mozilla/rhino: Rhino is an open-source implementation of JavaScript written entirely in Java," github.com, 2020. [Online]. Available: https://github.com/mozilla/rhino. [Accessed 05 03 2022].

[43] Falleri, F. Morandat, X. Blanc, M. Martinez and M. Monperrus, "Fine-grained and accurate source code differencing," *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, no. 12, p. 313–324, 2014.

[44] "Weka 3 - Data Mining with Open Source Machine Learning Software in Java," waikato, 2022. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/. [Accessed 24 03 2022].

[45] "Sample Size Calculator: Understanding Sample Sizes | SurveyMonkey," surveymonkey, 2022. [Online]. Available: https://www.surveymonkey.com/mp/sample-size-calculator/. [Accessed 01 05 2022].

[46] A. Saboury, P. Musavi, F. Khomh and G. Antoniol, "An empirical study of code smells in JavaScript projects," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 294-305, 2017.