Analyzing Bug Patterns in TypeScript: A Comparative Study of Static Typing's Impact on Modern Web Development

Justin Pope
Simon Fraser University
Canada
jbp3@sfu.ca

Maxwell Zhang
Simon Fraser University
Canada
mtz3@sfu.ca

Sadra Khorvash
Simon Fraser University
Canada
sma295@sfu.ca

***Abstract*—The adoption of TypeScript, a statically typed superset of JavaScript, has grown substantially in modern web development due to its promise of enhanced code reliability and maintainability. This study empirically examines bug patterns in TypeScript applications by analyzing over 15,000 issues from prominent repositories. Using manual labeling and automated classification, we identify the root causes and prevalence of bugs, comparing the results to previous studies on JavaScript and client-side applications. Our findings reveal significant differences in bug prevalence, such as a marked reduction in syntax errors and undefined methods due to TypeScript's static typing. However, challenges persist in runtime stability, configuration management, and performance optimization, reflecting the complexity of contemporary applications. By introducing new bug categories and uncovering trends in modern frameworks, this research provides actionable insights for developers and informs the future evolution of TypeScript-based tools and practices.**

***Index Terms*—TypeScript, JavaScript, Static Typing, Bug Analysis, Web Development, Angular, React.**

I. Introduction

A. Overview

The web development landscape has transformed significantly, evolving from static Web 1.0 pages to dynamic, interactive platforms of Web 2.0. This transition has been driven by technologies like JavaScript and frameworks such as Angular, React, and Vue, enabling rich user experiences and real-time interactions. As web applications grow in complexity, ensuring code reliability and maintainability has become a pressing challenge.

TypeScript, a statically typed superset of JavaScript, addresses these challenges by introducing static typing, interfaces, and advanced language features. [1] These capabilities catch errors at compile time, alledgedly improving code quality and reducing runtime issues. Its integration into modern frameworks like Angular and adoption by major repositories underscores its significance in contemporary software engineering.

There have been studies into the possible positive effects of using TypeScript over Javascript. "To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub" researched the effect of using Typescript over Javascript. To quantify code quality, the researchers used code smells and code understandability metrics from the static analysis tool "SonarQube". For software quality, researchers measured bug report frequency and bug resolution time. In terms of Typescript versus Javascript, the researchers found that Typescript projects score better in terms of code smell and understandability, but have equal or worse report frequency and bug resolution time than Javascript projects [2].

Despite TypeScript's benefits, there is limited research examining the specific types of bugs in TypeScript applications compared to their JavaScript counterparts. For instance, a 2013 study by Ocariza et al. on client-side JavaScript bugs found that 74% were due to invalid function parameters, with 65% involving the Document Object Model (DOM) [3]. A 2023 study focused on React applications updated this understanding but left a gap in exploring TypeScript-specific bugs [4].

Our research aims to fill this gap by conducting an empirical analysis of bugs in TypeScript repositories. We hypothesize that TypeScript's static typing reduces invalid function parameter errors and mitigates DOM-related issues. Our findings provide insights into how TypeScript features influence bug patterns and guide developers in improving practices for TypeScript-based frameworks.

B. Research Questions

To guide this study, we aim to answer the following research questions:

- **RQ1:** What root causes of bugs exist in TypeScript applications, and what is their prevalence?
- **RQ2:** How do TypeScript root causes compare to past studies on JavaScript applications?
- **RQ3:** What is the relative prevalence of bug symptoms in TypeScript applications?

Through answering these questions, this study seeks to provide a comprehensive understanding of the bug landscape in TypeScript applications. The findings will contribute to the broader knowledge of software reliability and guide developers in leveraging TypeScript more effectively to minimize common pitfalls.

II. Methodology

A. Manual Labeling

1) Repository Selection: Repositories were selected by going down the list of TypeScript-tagged repositories on GitHub, sorted by the number of stars. To qualify, a project had to consist of at least 50% TypeScript, according to GitHub, and be maintained by a primarily English-speaking developer team. The repositories studied are listed in table I

TABLE I
List of Repositories Used

Repository	Description
Angular	Libraries for the web development framework Angular
Angular-cli	cli tool for the web development framework Angular
TypeScript	Compiler for Typescript
VSCode	GUI code editor
Nest	Server side application framework
Storybook	Tool for building frontend UI
Strapi	Content management system with a GUI

2) Issue Eligibility: Issues from repositories were reviewed in random order to find bugs. To be considered, an issue had to be closed, involve Typescript or dependency management files, report a bug (not a suggestion), not be a duplicate of another issue, and have an accepted pull request or commit that fixed the bug.

3) Categorization: Issues were initially classified based on the categories described in the paper "An Empirical Study of Client-Side JavaScript Bugs" [3]. During labeling, the categories "configuration file" and "invalid mutator" were discovered and added. All labeled bugs were reviewed by two people to ensure correctness. The categories used are as follows:

- **Undefined/Null Variable Usage**: "A JavaScript variable that has a null or undefined value – either because the variable has not been defined or has not been assigned a value – is used to access an object property or method. Example: The variable x, which has not been defined in the JavaScript code, is used to access the property bar via x.bar [3]."
- **Undefined Method**: "A call is made in the JavaScript code to a method that has not been defined. Example: The undefined function foo() is called in the JavaScript code [3]."
- **Incorrect Method Parameter**: "An unexpected or invalid value is passed to a native JavaScript method, or assigned to a native JavaScript property. Example: A string value is passed to the JavaScript Date object's setDate() method, which expects an integer. [3]"
- **Incorrect Return Value**: "A user-defined method is returning an incorrect return value even though the parameter(s) is/are valid. Example: The user-defined method factorial(3) returns 2 instead of 6 [3]."
- **Syntax-Based Fault**: "There is a syntax error in the JavaScript code. Example: There is an unescaped apostrophe character in a string literal that is defined using single quotes [3]."
- **Configuration File**: The bug is caused by dependencies and is resolved by updating dependencies through "packages.json".
- **Incorrect Mutator**: A mutator method is incorrectly modifying the fields of its object.
- **Other**: "Errors that do not fall into the above categories. [3]"

B. Automatic Classification

1) Repository Selection and Data Collection: We analyzed a diverse set of TypeScript repositories with very high Typescript usage, averaging over 96.4 percent and including:

- **Angular** (87.8%): *angular/angular* - Web applications with confidence.
- **Formbricks** (90.2%): *formbricks/formbricks* - Open-source survey platform.
- **Zod** (96.2%): *colinhacks/zod* - Schema validation with static type inference.
- **NestJS** (99.9%): *nestjs/nest* - Enterprise-grade server-side applications.
- **Novu** (94.3%): *novuhq/novu* - Notification platform integrations.
- **Zustand** (98.8%): *pmndrs/zustand* - State management in React.
- **Prisma** (98.5%): *prisma/prisma* - Next-generation ORM for TypeScript.
- **Refine** (97.7%): *refinedev/refine* - React framework for internal tools.
- **Trigger.Dev** (99.2%): *triggerdotdev/trigger.dev* - Background jobs platform.
- **Type Challenges** (98%): *type-challenges/type-challenges* - TypeScript challenges.

Using a Python script, we collected issues from these repositories via GitHub's API. The script fetched issues and associated comments, ensuring comprehensive data coverage by handling pagination and rate limiting. This dataset focused on popular TypeScript-related bugs such as runtime errors, performance issues, compilation errors, and type system challenges.

2) Data Cleaning and Preprocessing: We applied a systematic data cleaning process to ensure relevance and clarity. Issues unrelated to TypeScript or written in non-English were removed. Keywords such as *type error*, *compiler*, *interface*, and *tsconfig* were used to filter issues. Using the `langdetect` library, we ensured that only English-language issues were retained. This process resulted in the removal of 11,006 issues (42% of the total data), streamlining the dataset for further analysis.

3) Bug Classification: To classify issues, we employed keyword-based matching. Fields such as issue titles, bodies, and comments were merged into a single text string, preprocessed to remove code blocks and extraneous symbols, and lemmatized for consistency. Using WordNet, we expanded keyword lists with synonyms to improve detection accuracy. Bugs were categorized into six primary groups:

1) **Runtime Errors**: Execution-time issues, including undefined functions and memory leaks.
2) **Performance Issues**: Inefficient code or resource bottlenecks.
3) **Compilation Errors**: Syntax and type-checking problems caught during compilation.
4) **Dependency Issues**: Missing or conflicting package dependencies.
5) **Invalid Function Parameters**: Incorrect or mismatched arguments in function calls.
6) **Type System Errors**: Issues with type mismatches, inference, or missing declarations.

Issues that did not match these categories were labeled as "Uncategorized."

III. Findings

A. *What is the relative prevalence of bug symptoms in Typescript applications?*

Our classification process analyzed 15,118 issues, successfully categorizing 90.05% (13,614 issues). The distribution is summarized in Table II.

TABLE II
Bug Category Distribution

Bug Category	Count	Percentage (%)
Runtime Errors	11,483	84.35
Performance Issues	579	4.25
Compilation Errors	541	3.97
Dependency Issues	395	2.90
Invalid Function Parameters	239	1.76
Type System Errors	377	2.77
Total Classified	13,614	90.05
Total Issues	15,118	100.00

The dominance of Runtime Errors (84.35%) highlights challenges in runtime stability and debugging, reflecting their direct visibility during application testing. While less frequent, Performance Issues (4.25%) and Compilation Errors (3.97%) are critical for application optimization and early-stage development. Dependency Issues, Invalid Function Parameters, and Type System Errors, though relatively rare, indicate areas where developers face challenges in leveraging TypeScript's static analysis and type safety effectively.

B. *What root causes of bugs exist in TypeScript applications and what is their prevalence?*

1) Relative Prevalence: The relative prevalence of bugs found in this paper is presented in figure 1. The counts of each category of issues by repository is given in table III. The results of the JS bugs paper are given in figure 2 as a comparison.

TABLE III
Issues in Repositories

Repository	o	irv	cf	im	imp	nvu	um
Angular	2	4	0	0	0	3	0
Angular-cli	10	3	6	0	1	0	0
TypeScript	3	3	0	1	3	0	0
VSCode	1	3	0	5	1	0	0
Nest	1	3	0	1	0	1	0
Storybook	9	0	1	0	0	0	1
Strapi	4	1	0	0	2	2	0
Total Classified	30	17	7	7	7	6	1

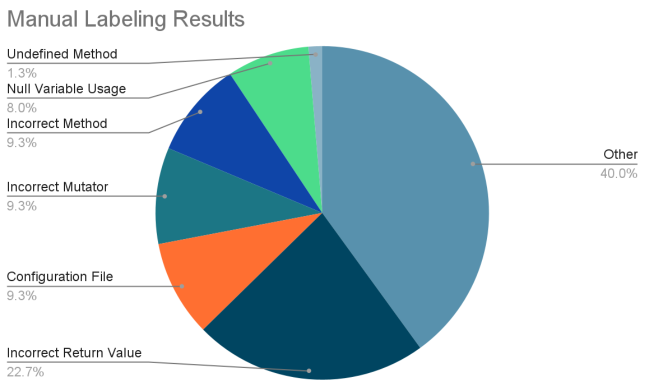

Fig. 1. Relative Prevalence of Bugs

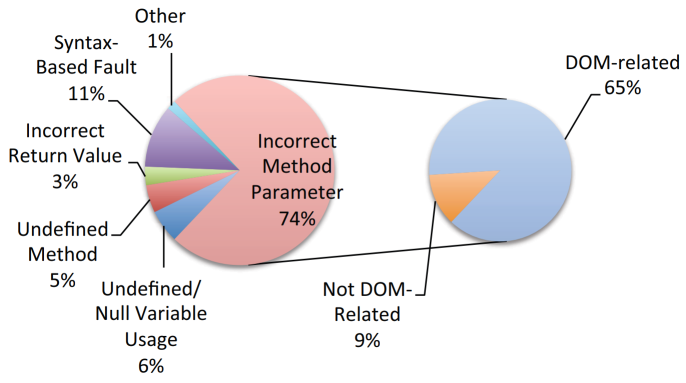

Fig. 2. Findings from [3]

2) New Categories: During classification, we introduced two new bug categories: "configuration file" and "incorrect mutator." Many bugs were resolved by updating dependencies in package.json, highlighting the widespread adoption of npm in modern development workflows. Additionally, we observed that a significant number of bugs were related to mutator methods. This trend may reflect the increased use of object-oriented programming in modern applications compared to JavaScript applications from 2013.

C. *How do TypeScript root causes compare to past studies on JavaScript applications?*

1) Invalid Method Parameter: The most significant difference between our results and those of the JS Bugs paper is the prevalence of invalid method parameter bugs. The JS Bugs paper reported that 74% of bugs were due to invalid method parameters, with 88% of those being DOM-related [3]. In contrast, our study found that invalid method parameter bugs accounted for only 8.33% of all bugs. This discrepancy may be attributed to the differing types of applications analyzed. The JS Bugs paper focused on eight web applications and four libraries, whereas four out of the seven repositories in our study had no significant GUI component.

Additionally, we did not track whether bugs were DOM-related. However, only 3/7 invalid method parameter bugs identified in our study originated from applications with GUIs. The way GUIs are developed today also differs. Notably, two of the three GUI projects analyzed in our study used React, which could have multiple implications. Features such as IDE support and React's compilation process may reduce the likelihood of invalid method parameter bugs. Furthermore, the functional approach React uses to build GUI behavior means that UI-related bugs may be more likely classified as invalid return value issues. Overall, DOM-related invalid method parameter bugs are far less prevalent in modern TypeScript applications.

2) Other: The "Other" category of bugs was significantly more prevalent in our study, accounting for 40% of all issues, compared to just 1% in the JS Bugs paper [3]. This is despite the addition of new categories. This substantial increase in "Other" bugs could reflect the greater complexity of modern applications compared to client-side JavaScript applications from 2013. Many bugs in contemporary TypeScript applications do not fit neatly into simple categories.

3) Invalid Return Value: Invalid return value bugs accounted for 3% of issues in the JS Bugs paper but represented 22.67% in our study [3]. This disparity could be due to the increased complexity of the applications analyzed. Functions in modern applications could be more complicated and therefore more prone to errors.

4) Null Variable Usage: Null value usage bugs accounted for 6% of bugs in our study, compared to 8% in the previous study [3]. This result is somewhat surprising, as one might expect TypeScript's null safety features to reduce their frequency. However, it is noteworthy that their prevalence, relative to categories like invalid return value bugs, is lower in our findings than the JS bugs paper. This is worth noting because the results of the JS bugs paper were dominated by DOM-related invalid method parameter bugs.

5) Undefined Method: Bugs rooted in undefined methods accounted for 5% of issues in the JS Bugs paper [3], but only 1.33% in our study (a single bug). This bug arose because a required built-in method was missing from certain older browser versions. While one might expect TypeScript's type system to prevent undefined method bugs entirely, our findings suggest that it is highly effective in minimizing such errors.

6) Syntax Based Fault: No syntax-based faults were identified in our study, compared to 11% in the JS Bugs paper [3]. This result aligns with expectations, as syntax errors in TypeScript typically result in compilation failures, preventing them from being introduced into production code.

IV. Threats to Validity

A. Automatic Classification

This study's validity faces several potential threats. Internally, the reliance on keyword-based classification may result in misclassifications or missed issues, and automated filtering lacks the nuance of human judgment. Externally, focusing on a limited set of high-TypeScript-usage repositories may not fully represent the broader ecosystem, especially smaller or mixed-language projects. Construct validity could be affected by the chosen bug categories, which may oversimplify complex issues or overlap while excluding "Uncategorized" issues omits potentially meaningful data. Lastly, reliance on GitHub's API and data from a specific time frame may introduce gaps or outdated insights, limiting reliability and generalizability.

B. Manual Labeling

Manual labeling introduces several potential threats to the validity of our findings, which we sought to mitigate as much as possible:

- **Subjectivity in Classification:** Bug classification involves subjective judgment. To reduce bias, two reviewers analyzed each issue, and only consensus-labeled cases were included.
- **Bias in Category Assignment:** Newly introduced categories like "Configuration File" and "Incorrect Mutator" risk overclassification. A retrospective review ensured consistent labeling across the dataset.
- **Incomplete Understanding:** Misclassification risks arise from incomplete or ambiguous issue descriptions, comments, or pull requests. Thorough analysis mitigated this, though some errors may persist.
- **Sample Representativeness:** The most starred TypeScript repositories on Github may not be generalizable to Typescript projects as a whole. Random issue selection aimed for diversity but may not fully capture all bug types. Focusing on closed, non-duplicate bugs with fixes may exclude recurring patterns. Bugs labeled per repository wasn't consistent.
- **Researcher Fatigue:** Manual labeling is labor-intensive, potentially leading to errors. Dividing tasks among researchers helped mitigate fatigue.
- **Repository Evolution:** Changes in issue reporting and bug-fixing practices over time may affect patterns, especially for older issues.

While these threats are inherent to manual labeling methodologies, our mitigation strategies aimed to reduce their impact and ensure the validity of our findings.

V. Conclusion

This study offers a detailed exploration of bug patterns in TypeScript applications, highlighting the impact of static typing on modern web development. Our findings demonstrate that while TypeScript effectively minimizes certain classes of bugs, such as syntax errors and undefined methods, it also reveals areas where challenges persist, such as runtime stability and performance optimization. The stark differences in bug prevalence compared to prior JavaScript studies underscore the evolving nature of web application development, influenced by changes in programming paradigms, frameworks, and tooling.

The introduction of new bug categories, like configuration file and incorrect mutator issues, reflects the growing complexity of modern applications. These insights not only illuminate the current bug landscape but also serve as a roadmap for developers seeking to leverage TypeScript's features effectively.

Future work could explore the interplay between TypeScript and newer frameworks, delve into the role of specific IDE features, or investigate long-term trends in bug resolution efficiency. By addressing the challenges identified in this study and capitalizing on TypeScript's strengths, developers and tool creators can further enhance the reliability, maintainability, and overall quality of web applications built with TypeScript.

References

[1] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *ECOOP 2014 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, R. Jones, Ed., vol. 8586. Berlin, Heidelberg: Springer, 2014, pp. 257–281, https://doi.org/10.1007/978-3-662-44202-9$_1$1.

[2] J. Bogner and M. Merkel, "To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 658–669.

[3] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 55–64.

[4] W. Luxi and S. Yuxin, "An empirical study comparing javascript bugs in 2013 and 2023," 2023.

VI. Appendix

Scripts used in our study and data for the automatic classification can be found in this repository: https://github.com/sadramanouch/479Proj

The data for manual classification can be found here: https://drive.google.com/file/d/18xGMTO8r_kKSQeTLkvmDuzIXkYhvTmpx/view?usp=drive_link