Journal of Software Engineering Research and Development, 2026, 14:12, doi: 10.5753/jserd.2026.5985

How TypeScript Compiler Options Influence Code Quality: A Correlation Study with Git Repositories

Thomaz Zandonotto [**Unisinos** | ***thomaz.zandonotto@gmail.com***]
Alex Roehrs[**Unisinos** | ***alexr@unisinos.br***]
Kleinner Silva Farias de Oliveira[**Unisinos** | ***kleinnerfarias@unisinos.br***]
Jorge Luis Victória Barbosa[**Unisinos** | ***jbarbosa@unisinos.br***]
Cristiano André da Costa[**Unisinos** | ***cac@unisinos.br***]
Rodrigo da Rosa Righi[**Unisinos** | ***rrrighi@unisinos.br***]

Abstract *TypeScript* is widely adopted in modern software development to improve code reliability through static typing. However, its effectiveness in promoting code quality depends heavily on the compiler options configured within each project. Despite its popularity, little is known about how these compiler configurations influence code quality in practice. This study investigates the relationship between *TypeScript* compiler options and code quality metrics in real-world software projects. We analyzed over 4,000 public *GitHub* repositories that use *TypeScript*, through an automated four-stage pipeline comprising data mining, compiler configuration extraction, static quality assessment, and statistical correlation analysis. Projects were selected based on predefined criteria, and their `tsconfig.json` files were parsed to extract compiler settings. *SonarQube* was employed to evaluate bugs and code smells, aggregated into a composite Static Issues Index (SII), which was then correlated with specific compiler options. The results show that stricter compiler settings, such as `noImplicitAny` and `noUnusedParameters`, tend to correlate with fewer bugs and code smells. However, some options behave counterintuitively at scale, for instance, enabling `noFallthroughCasesInSwitch` and `noUncheckedIndexedAccess` correlated positively with bug counts in our dataset, suggesting the presence of confounding factors rather than a direct causal effect. These findings provide large-scale empirical evidence that compiler configurations in *TypeScript* are associated with measurable differences in static code quality. The study offers practical guidance for developers seeking to enforce best practices through *TypeScript*'s configuration mechanisms and contributes an empirical perspective to the software engineering community.

Keywords: *JavaScript, TypeScript, Compiler Configuration, Code Quality, Static Analysis, GitHub Mining*

1 Introduction

During the past ten years, the software industry has witnessed the consolidation of *JavaScript* as a widely adopted programming language used in a variety of contexts (Wang et al., 2025). For the 13th consecutive year, *JavaScript* was cited in 2024 as the most widely used programming language (Stack Overflow, 2024). With the emergence of new tools and technologies, *JavaScript* has evolved into a flexible technology that is now used in many different types of projects and contexts, powering highly complex applications on both the client side and the server side (Wirfs-Brock and Eich, 2020).

Despite its simplicity and adaptability, which have contributed to *JavaScript*'s rapid adoption in recent years, many of the language's characteristics can ultimately reduce the quality of the software produced, making it challenging to use in large and more complex codebases (Desmazières et al., 2025; Fard and Mesbah, 2013). Previous empirical studies have shown that *JavaScript* projects often suffer from maintainability issues and higher defect rates due to its dynamic typing and lack of compile-time checks (Hu et al., 2025; Wang et al., 2025; Bogner and Merkel, 2022; Gao et al., 2017). *TypeScript* was created to address many of the issues introduced by the widespread use of *JavaScript*, offering features that facilitate large-scale software development (Wang et al., 2025; Black, 2020).

In recent years, *TypeScript* has gained significant popularity among developers, with adoption increasing in both small-scale and large-scale projects, from 12% in 2017 to 37% in 2024 (JetBrains, 2024). Empirical evidence suggests that the adoption of *TypeScript* and static typing mechanisms can significantly reduce defect density and improve code maintainability compared to *JavaScript* (Wang et al., 2025; Bogner and Merkel, 2022).

However, the degree to which strictness is adopted in *TypeScript* projects depends on configuration: *TypeScript* can behave more dynamically or more statically depending on compiler options (TypeScript Project, 2025), allowing teams to introduce type features incrementally per project (Prescott, 2018). We therefore hypothesize that the degree of strictness selected via compiler options correlates with measurable differences in code quality across real-world projects.

This work empirically examines the practical effects of *TypeScript* compiler settings on code quality in real-world projects. Unlike prior work that centered on the theoretical benefits of static typing, the advantages of *TypeScript* over *JavaScript*, or the concept of gradual typing, this study focuses on concrete compiler options to provide a detailed, applied, large-scale view of how configuration choices impact software quality in practice.

In the following paragraphs, examples are provided to frame the research problem addressed in this study and to illustrate the potential consequences that may arise from different *TypeScript* compiler options. These examples illustrate how different compiler options can prevent runtime failures, directly affecting software reliability and correctness.

The code in Figure 1 exemplifies a variable `text` that explicitly holds either a `string` or a `null` value. In this example, when the `strictNullChecks` compiler option is not enabled, passing `null` as an argument to the `printStringLength` function will result in a runtime error. This error occurs because the function attempts to access the `length` property, which is `undefined`. The *TypeScript* compiler can prevent this runtime error when `strictNullChecks` is explicitly enabled.

```typescript
function printStringLenght(s: string) {
  console.log(s.length)
}

let text: string | null = null;

printStringLength(text)
```

Figure 1. `strictNullChecks` usage example

Somewhat similar to the `strictNullChecks` configuration is the `noImplicitAny` option, which helps prevent the unintended use of the `any` type. Figure 2 illustrates the `sum` function, which adds two numbers.

```typescript
function sum(a, b) {
  return a + b;
}

sum(5, "10");
```

Figure 2. `noImplicitAny` usage example

In the example, the parameter types are not explicitly declared as numbers and are implicitly typed as `any` by the compiler. At runtime, *JavaScript* performs implicit type coercion and treats the `number` as a `string`, resulting in the concatenation of values instead of the intended numeric addition. The program outputs “510”, which is an unexpected result. This error could have been caught at compile time if `noImplicitAny` had been explicitly enabled, indicating the type of mismatch between the parameters.

Additionally, Figure 3 shows the effect of using the `noUncheckedIndexedAccess` compiler option, which helps prevent run-time errors by imposing explicit checks when accessing indexed properties.

```typescript
const colors: string[] = ["white", "blue", "yellow"];
const index: number = 3;

console.log(myArray[index].toUpperCase());
```

Figure 3. `noUncheckedIndexedAccess` usage example

When this compiler option is turned off, the code will fail due to a run-time error when trying to call the `toUpperCase` method. This occurs because the `colors` array does not have a value at index 3. By explicitly enabling this option, the *TypeScript* compiler requires that indexed accesses be explicitly checked for `undefined` or `null` values.

As shown in the examples above, the *TypeScript* compiler’s default behavior is relatively lenient, favoring flexibility over strictness. This flexibility, designed to ease the rapid prototyping and migration of existing *JavaScript* codebases, means that stricter options are not enforced by default. We postulate that this characteristic may not provide the level of type safety often desired or necessary in complex *TypeScript* projects, potentially leading to runtime errors if not properly configured.

Motivated by these observations, this study examines the relationship between *TypeScript* compiler options and code quality in real-world *GitHub* projects, electing the following research question as the primary one to guide this investigation:

How do different TypeScript compiler options correlate with code quality?

In addition to the correlation analysis that directly addresses the research question, the study also includes exploratory analyses of descriptive statistics and configuration usage. These intermediate steps aim to characterize the dataset and provide context for interpreting the correlation results, such as understanding how bugs, code smells, and compiler options are distributed across the analyzed projects. This descriptive evidence supports the robustness of the subsequent statistical inferences.

Understanding this relationship is essential for software engineers aiming to improve the reliability and maintainability of large-scale *TypeScript* systems. By addressing this question and analyzing its implications, this study provides empirical evidence on how specific compiler configurations influence observable quality metrics, supporting practitioners in selecting compiler options that reduce defects and improve code maintainability.

The scientific contributions of this research are threefold and aim to advance the field of software engineering by offering both methodological and practical insights:

1. **Empirical:** We conducted a large-scale correlation analysis involving more than 4,000 real-world *TypeScript* repositories, identifying how specific compiler options relate to code quality metrics.
2. **Methodological:** We propose a robust and fully automated pipeline based on *GitHub* mining, *SonarQube* static analysis, and statistical evaluation that researchers and practitioners can reuse or adapt for similar studies in different contexts.
3. **Practical:** We provide an overview of how compiler options are adopted across open-source projects, identifying configuration patterns that correlate with higher or lower code quality. This analysis guides developers in selecting effective compiler options that may prevent problems and improve maintainability.

The remainder of this article is organized as follows. Section 2 (*Background*) presents the foundational concepts of this study. Section 3 (*Related Work*) reviews previous relevant research. Section 4 (*Material*) describes the dataset and materials used. Section 5 (*Method*) outlines the methodologies employed. Section 6 (*Results and Discussion*) presents and discusses the research findings. Section 7 (*Threats to Validity*) discusses the limitations and potential biases of the study. Finally, Section 8 (*Conclusion*) concludes the article and proposes directions for future work.

2 Background

This section provides the technical and conceptual background necessary to understand the research problem proposed by this work. The illustrative examples presented in Section 1 (Figures 1–3) show how compiler options influence code behavior; this section formalizes those intuitions by defining code quality and static analysis concepts.

Code Quality in Software Engineering is a multifaceted concept, capturing how well software meets functional and non-functional requirements. A possible way is to subdivide quality into internal and external facets. Internal quality refers to intrinsic code properties, such as readability, modularity, and complexity (Oliveira et al., 2022; Fernandes et al., 2020). External quality, on the other hand, reflects how well the software meets user needs and expectations (International Organization for Standardization and International Electrotechnical Commission, 2024).

Static Analysis refers to the process of examining source code without executing it, often to identify code quality concerns (Ayewah et al., 2008). It involves the use of automated tools such as *SonarQube* to analyze the structure, syntax, and semantics of the source code, allowing the identification of defects and violations of coding standards (Chess and McGraw, 2004). Static code analysis operates at a static level, meaning it inspects the code as written, independently of its runtime behavior.

In the context of this study, the notion of code quality is exercised through the diagnostic taxonomy implemented by *SonarQube*, a static analysis tool that maps assessments to the *ISO/IEC 25010* quality characteristics and reports findings as bugs and code smells (SonarSource, 2025). Aligning our measurement approach with this model ensures methodological consistency and reproducibility across real-world projects.

JavaScript is a programming language created by Brendan Eich in the mid-1990s (International, 2025). It has since become a cornerstone of both web and application development (W3Techs, 2025). As a dynamic, high-level, interpreted language, *JavaScript* was originally developed for use in web browsers. However, its utility has expanded significantly with the introduction of server-side platforms such as NodeJS, enabling the development of server-side applications (Tilkov and Vinoski, 2010).

TypeScript is a statically-typed superset of *JavaScript* introduced by Microsoft in 2012. By providing a robust type system, *TypeScript* helps developers catch potential bugs through compile-time checks, thus reducing the likelihood of run-time errors. Its compatibility with existing *JavaScript* allows developers to gradually introduce static typing and leverage the features of *TypeScript* in existing projects (Goldberg, 2022). With its focus on developer productivity and code maintainability, *TypeScript* has emerged as a valuable language choice for large-scale software running in *JavaScript* runtime environments, such as browsers and NodeJS. It provides a solid foundation for building robust, scalable, and maintainable applications (Bierman et al., 2014).

The idea of a *gradual type system* refers to type systems that allow seamless integration of statically typed and dynamically typed code within the same programming language. Languages that implement this concept provide a way to combine the benefits of static typing, such as early error detection, improved tooling, and better maintainability, with the flexibility and convenience of dynamic typing, including runtime adaptability, rapid prototyping, and easier code evolution. (New et al., 2021).

Git is a distributed version control system that plays a fundamental role in the development of software projects. It enables developers to track changes to the codebase over time, facilitating collaboration and maintaining a robust history of code modifications. With Git, developers can create branches, merge changes, and revert to previous versions, ensuring a smooth and controlled development process (Chacon and Straub, 2014).

GitHub is a popular web-based hosting service built around Git. It provides a platform for developers to share their projects, collaborate with others, and contribute to open source software. *GitHub* improves Git's functionality by offering features such as issue tracking, pull requests, and code reviews, fostering a collaborative ecosystem for software development (GitHub, Inc., 2025).

3 Related Work

We conducted a literature review using Google Scholar, IEEE Xplore, ACM Digital Library, and SpringerLink. The review analyzed studies published within the past two years on the aforementioned platforms. The main search terms used to find relevant articles included: "*TypeScript compiler options*", "*JavaScript vs TypeScript*", "*software quality*", "*JavaScript migration to TypeScript*", "*TypeScript feature adoption*", "*type theory*" and "*gradual typing theory*".

Initially, 31 articles were retrieved from the research portals using the defined search terms. These articles were first filtered by title, reducing the number to 19. A subsequent abstract-based screening narrowed the selection to 7 articles, and a full-text review resulted in 4 selected articles included in the final analysis.

Despite no prior studies specifically addressing the impact of different *TypeScript* compiler options on code quality, several related works offer a foundational understanding of programming languages, static typing, and software quality, which are relevant to our research.

The article *Safe & Efficient Gradual Typing for TypeScript* (Rastogi et al., 2015) explores the implementation and benefits of gradual typing in *TypeScript*. The primary research question investigates how gradual typing can be integrated into *TypeScript* both effectively and safely, without compromising performance. The authors propose a type system that enables a seamless blend of dynamic and static typing, facilitating smooth transitions between the two. Their findings are highly relevant to our research, as they show that gradual typing can improve both code safety and performance. This suggests that appropriately configuring *TypeScript* compiler options could help harness these benefits to improve code quality.

The article *To Type or Not To Type: Quantifying Detectable Bugs in JavaScript* (Gao et al., 2017) examines the effective-

Table 1. Comparison of related empirical studies and this work

Study	Scope / Focus	Metrics Used	Granularity of Analysis	Empirical Method
(Rastogi et al., 2015)	Implementation of gradual typing in TypeScript	Type soundness, performance, and safety	Type system design	Experimental and formal analysis
(Gao et al., 2017)	Detectable bugs in JavaScript with and without type annotations	Bug frequency	Language-level (JavaScript vs. typed JS)	Large-scale mining of GitHub repositories
(Ray et al., 2017)	Relationship between programming language choice and code quality	Bug and defect density across multiple languages	Cross-language comparison	Mining open-source projects and statistical correlation
(Bogner and Merkel, 2022)	Software quality comparison between JavaScript and TypeScript	Maintainability, code smells, cognitive complexity, and bug counts	Language-level (TypeScript vs. JavaScript)	Static analysis of open-source projects (SonarQube)
This study	Impact of TypeScript compiler configurations on code quality	Bugs, code smells, and composite issue index (SII)	Compiler-level configuration granularity	Automated mining, static analysis (SonarQube), and statistical correlation

ness of type annotations in *JavaScript* by comparing the frequency of detectable bugs in typed versus untyped code. The research question investigates whether the addition of type annotations reduces the number of bugs in *JavaScript* code. The authors concluded that static typing, as implemented in *TypeScript*, can significantly decrease the occurrence of certain types of bugs. This study highlights the potential benefits of static typing, emphasizing how *TypeScript*' type system can help prevent common programming errors. Their findings support the hypothesis that stricter typing rules, enforced through *TypeScript* compiler options, can enhance code reliability and reduce the incidence of errors.

The article *A Large Scale Study of Programming Languages and Code Quality* (Ray et al., 2017) investigates the correlation between programming languages and code quality in a wide range of open-source projects. The primary research question explores whether the choice of programming language significantly affects the quality of the code. The authors found that, while language choice does influence code quality, it is not the sole determining factor. This work is relevant because it highlights the broader relationship between language features and software quality, establishing a foundation for exploring more specific aspects, such as *TypeScript* compiler options. Their findings suggest that language characteristics can affect quality, although other factors also play a significant role.

The article *To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub* (Bogner and Merkel, 2022) systematically compares *JavaScript* and *TypeScript* applications in terms of software quality. The primary research question investigates whether *TypeScript* applications demonstrate better quality metrics than their *JavaScript* counterparts. The authors found that *TypeScript* projects generally have fewer bugs and exhibit better maintainability. This finding is directly relevant to our investigation, as it suggests that leveraging *TypeScript*' compiler options could further enhance these quality metrics. Their comparison highlights how the static typing and compiler features of *TypeScript* contribute to overall improvements in software quality.

More recent investigations have also expanded the empirical understanding of *TypeScript* and static analysis in modern contexts. For example, (Hu et al., 2025) examined the suppression and evolution of static analysis warnings in large-scale projects, highlighting how developers' interaction with tooling affects code quality outcomes.

Similarly, (Wang et al., 2025) conducted a large-scale study on *TypeScript*-specific bugs, providing complementary information on how typing mechanisms relate to reliability. Together with recent language evolution analyses (Desmazières et al., 2025) and updated ISO/IEC 25002:2024 (International Organization for Standardization and International Electrotechnical Commission, 2024) standards for software quality assessment, these works reinforce the timeliness of the present study, which focuses on compiler-level configurations as an emerging determinant of quality in modern static typing ecosystems.

Building on recent empirical advances in static analysis and *TypeScript* research (e.g., Hu et al., 2025; Wang et al., 2025; Desmazières et al., 2025), this study further expands the current understanding of how language mechanisms and configuration practices influence software quality. To better contextualize how this work differs from previous research, Table 1 summarizes the scope, metrics, and empirical methods of key related studies.

As shown in Table 1, previous studies have provided valuable information on the relationship between programming languages, type systems, and software quality. In contrast, this study advances the state of the art by empirically analyzing *TypeScript*' compiler configuration granularity, provid-

ing evidence on how individual compiler options correlate with static quality metrics in real-world projects.

4 Material

In this section, we present the core components of our research, beginning with the objective of analysis, which involves using the *GitHub* API to access and extract *TypeScript* code samples from various repositories. We then detail the criteria used to select and qualify *GitHub* repositories for inclusion in this study, ensuring both relevance and quality. Finally, we outline the quality metrics considered and the *TypeScript* compiler options identified as relevant to this investigation.

4.1 Object of Analysis

The core material of this research is the *GitHub* API, which enables programmatic access to the vast collection of code repositories hosted on *GitHub*. It offers powerful means for extracting data at scale, allowing us to compile a large set of *TypeScript* code samples for analysis. Using the *GitHub* API, we were able to gather *TypeScript* code samples that serve as the basis for our study (Bonino Quintana and Fagerlind, 2020).

4.2 Qualifying *Git* Repositories

We required *GitHub* projects to meet seven criteria to ensure the relevance, high quality, and representativeness of the target population, as follows.

1. The project must be a public repository, ensuring it is accessible both to the general public and through the *GitHub* API.
2. The project must not be a fork of another *GitHub* repository, as forks are copies of existing projects and may not represent original work.
3. The project must not be a *GitHub* template, as templates are intended to serve as starting points for new projects and may introduce boilerplate code into our analysis.
4. The project must have more than 5 *GitHub* stars, indicating a minimum level of community interest and engagement.
5. The project must have *TypeScript* as its primary programming language. This ensures that the analysis focuses on projects written in the language relevant to our research.
6. The project must contain a *tsconfig.json* file in its root directory. This simplifies the analysis by ensuring that *TypeScript* configurations are centralized, avoiding cases where the settings are scattered across multiple files or use unconventional configurations.
7. The project's *tsconfig.json* file must not be extended to another configuration file. This also simplifies the analysis, as resolving inherited configuration options can be challenging, especially when they reference external packages.

4.3 Observed Quality Metrics

Following the conceptual framework introduced in Section 2, the quality metrics analyzed in this work were selected according to the diagnostic dimensions defined by *SonarQube*. Rather than adopting an arbitrary or purely canonical definition of "code quality," this study aligns with the conceptual taxonomy employed by the tool itself, which distinguishes three main categories of findings: *bugs*, *code smells*, and *vulnerabilities*. Since vulnerability detection is available only in commercial editions of *SonarQube*, this work focuses on the two quality-related dimensions exposed by the Community Edition: *bugs* and *code smells*.

In the *SonarQube* model, *bugs* represent source code defects with a high probability of causing incorrect behavior or runtime errors, whereas *code smells* indicate maintainability issues that may reduce readability, modularity, or extensibility of the software. Both dimensions are widely adopted in empirical software engineering as proxies for internal quality attributes such as reliability and maintainability (Ayewah et al., 2008; Fernandes et al., 2020; Bogner and Merkel, 2022). Therefore, these two metrics were selected to capture complementary aspects of static quality within the analyzed projects.

To support aggregate analyses, we computed a composite indicator we named *issues*, which corresponds to the total count of *SonarQube* findings classified as bugs or code smells. This aggregation allows correlation analyses to be performed on a single normalized metric while preserving coherence within the conceptual framework of *SonarQube*.

All metrics were extracted directly from *SonarQube* version 10.6 using the default *TypeScript* quality profile, and subsequently normalized by thousands of lines of code (KLOC, also known as Kilo Lines of Code) to allow comparisons between repositories of different sizes. As the community edition does not provide severity-weighted counts, all findings were treated uniformly in this study. Nevertheless, *SonarQube*' internal classification ensures methodological consistency across all projects, enabling reliable relative comparisons and replication.

4.4 Relevant Compiler Options

TypeScript compiler options include a wide range of configurations: from controlling code emission to syntax checking and enforcing strict type safety. The criteria for selecting the compiler options in this study were based on their documented impact on static typing and code style, as specified in the official *TypeScript* documentation (TypeScript Project, 2025). Although the compiler provides more than a hundred configuration options, only a subset directly affects static analysis and typing behavior, which are the focus of this research.

The compiler options considered in this study were selected from the official *TypeScript* 5.4 documentation (TypeScript Project, 2025). Repositories that used earlier versions were still included as a result of the data mining process. In such cases, any unsupported compiler options were simply treated as unused. In addition, we observed that the compiler options introduced after version 5.4, up to version 5.8, do

not meet the criteria aforementioned and therefore do not alter the scope or conclusions of this study (TypeScript Project, 2025). This reinforces the continued relevance and validity of the findings, even in light of the newer *TypeScript* versions. The compiler options chosen for this study are as follows:

1. **allowUnreachableCode**: Whether the compiler should allow unreachable code.
2. **allowUnusedLabels**: Allows for the use of labels that are declared but not referenced.
3. **alwaysStrict**: Enforces JavaScript strict mode (”use strict”) in every file, automatically adding the directive at the top of the compiled output.
4. **exactOptionalPropertyTypes**: Treats optional property keys as distinct from `undefined`, requiring `undefined` to be explicitly included in their type if needed.
5. **noFallthroughCasesInSwitch**: Prevents fall-through behavior between `case` clauses in `switch` statements.
6. **noImplicitAny**: Requires explicit type declarations, preventing the implicit use of `any`.
7. **noImplicitOverride**: Ensures that the `override` keyword is explicitly used when overriding a method from a base class.
8. **noImplicitReturns**: Ensures that all code paths within a function return a value.
9. **noImplicitThis**: Requires explicit type annotations for `this` keyword.
10. **noPropertyAccessFromIndexSignature**: Prevents property access using index signatures unless explicitly defined.
11. **noUncheckedIndexedAccess**: Ensures that indexed property accesses are properly type-checked.
12. **noUnusedLocals**: Flags variables that are declared but never used.
13. **noUnusedParameters**: Flags function parameters that are declared but never used.
14. **strict**: Enables all strict type-checking options in the compiler.
15. **strictBindCallApply**: Ensures correct type-checking for the `bind`, `call`, and `apply` methods.
16. **strictFunctionTypes**: Ensures that function-type signatures are strictly checked.
17. **strictNullChecks**: Prevents assigning `null` or `undefined` to values of non-nullable types.
18. **strictPropertyInitialization**: Ensures that all class properties are properly initialized before use.
19. **useUnknownInCatchVariables**: Requires variables in `catch` clauses to be typed as `unknown`.

Although the *TypeScript* compiler exposes more than one hundred configuration options, many of them concern build paths, module resolution, library inclusion, or transpilation targets, dimensions that do not affect static typing or code quality analysis. Therefore, the present study deliberately focused on the 19 options that directly influence static analysis behavior and type safety. This methodological boundary ensures that the correlations reported here are interpretable in terms of code quality rather than compilation or deployment artifacts. Consequently, the scope of the study is intentionally narrow and based on theory, prioritizing internal validity over exhaustive coverage.

5 Method

This section outlines the methodology used to carry out this research. The investigation of how different *TypeScript* compiler options correlate with code quality was carried out through four sequential and interdependent stages: (A) 5.1 Data Mining , (B) 5.2 Compiler Options Assessment, (C) 5.3 Quality Assessment and (D) 5.4 Statistic Analysis. The first three stages were designed to be executed through software automation using *TypeScript* and *Node.js*. The final stage was performed through direct analysis of the data obtained from the previous steps, using *Python* scripts written with libraries such as *Pandas*, *Matplotlib*, *Seaborn*, and *Scikit-learn*.

5.1 Data Mining

This phase aimed to identify *GitHub* projects that use *TypeScript* as their primary programming language and meet the specified criteria. A custom program iterated the results obtained from the *GitHub* API and stored the relevant project metadata. Figure 4 illustrates this process, which is described below:

A1 We start by querying the *GitHub API* to fetch a paginated list of *TypeScript* repositories. The query used ensures that the projects meet criteria 1, 2, 3, 4, and 5 as described in Section 4. The code snippet in Figure 5 was extracted from the implemented program and demonstrates how *GitHub* API requests are performed.

A2 After querying the repositories, we filtered out those that had already been processed in previous iterations. As we cannot ensure that the *GitHub* API will not return the same repository more than once, this precautionary step prevents redundant entries from being created in the dataset.

A3 With the filtered list of projects, the system then requests project-specific data for each remaining entry. Individual requests were sent to the *GitHub* API to collect pertinent information, such as the number of *GitHub* stars and other relevant metadata. This step enables the system to collect the complete data required for further analysis.

A4 After retrieving project-specific data, the program applies an additional filtering step to exclude projects that did not meet the study criteria. Projects that did not qualify, based on predefined conditions, were discarded and removed from the results.

A5 Following the filtering process, the program stores the project metadata collected in a *MongoDB* database. This step involved creating records and mapping the project information to the appropriate collection. By organizing the data in a structured manner, the system facilitated easy retrieval, analysis, and future reference.

A6 To continue the process, the system queries the *GitHub* API again, specifically targeting the next page of *TypeScript* repositories and continuing to iterate through the paginated results.

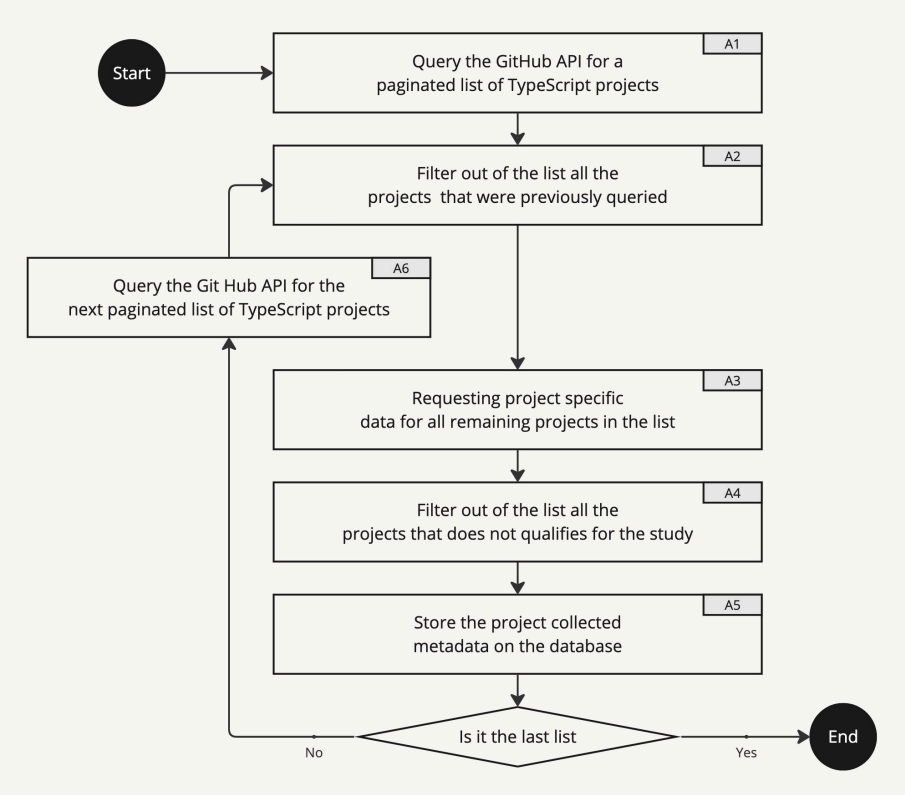

Figure 4. Data Mining Diagram

```typescript
export async function getRepositories(date: Date, page: number = 1) {

  const [ startDate, endDate ] = utils.getDailyDateRange(date)

  const start = startDate.toISOString()

  const end = endDate.toISOString()

  const apiQuery =
    + `q=language:typescript`
    + `+stars:>5`
    + `+created:${start}..${end}`
    + `+template:false`
    + `+public:true`
    + `+fork:false`

  const requestQuery =
    + `&page=${page}`
    + `&order=desc`
    + `&sort=stars`
    + `&per_page=30`

  const requestUrl =
    + 'https://api.github.com/'
    + 'search/repositories?'
    + apiQuery
    + requestQuery

  const { data, status } = await axios.get(requestUrl, { headers: {
    Authorization: `token ${GITHUB_TOKEN}`,
    Accept: GITHUB_ACCEPT_HEADER,
  }})

  if(status !== 200) {
    throw new Error('Error fetching Github repositories')
  }

  return parseRawGithubRepositoryDataArray(data.items)
}
```

Figure 5. GitHub Data Mining Code Snippet

A crucial aspect of the data mining process involves addressing the limitation imposed by the *GitHub* API, which restricts results to a maximum of 100 pages per query (GitHub, Inc., 2025). This constraint prevents exhaustive iteration over all available *TypeScript* repositories using a single query. To avoid this, we divided the queries into smaller one-day intervals and executed multiple requests, each scoped to retrieve repositories created on a specific day.

The data collection process was conducted over a period of ten business days. Each nightly execution of the scrapper ran for approximately eight hours, processing a different batch of repositories per iteration. At the beginning of each subsequent day, a manual sanity check was performed to validate the integrity and completeness of the extracted data before preparing the next batch for processing.

5.2 Compiler Options Assessment

This step involved scanning the repository files to detect the presence of a `tsconfig.json` file, accessing it, and extracting the relevant data required to perform correlation analysis between *TypeScript* usage and quality metrics. This section is divided into five sequential steps, each executed for every project. These steps are as follows:

- B1 We started by retrieving the identified *GitHub* project and downloading it to a local directory. This step involved accessing the project repository URL, initiating the cloning process using the *Git* CLI, and saving the project files locally.
- B2 Once the download was completed, we verified the eligibility of the project by checking for a `tsconfig.json` file in the project root directory. This ensured that criterion 6, as mentioned in Section 4, was

met. If the project did not contain this file, the execution was halted and no additional data was collected.

B3 We evaluated whether the configuration file was an extension of another configuration. This ensured that criterion 7, as described in Section 4, was met. If the project was found to extend another configuration or did not contain the required file, the execution was halted, and no additional data were collected.

B4 Finally, after all necessary checks were completed, we parsed the identified configuration file into a consistent data format. This step considered not only the explicitly declared configurations but also the default values used by the compiler. The resulting data format was a boolean representation that indicates whether a given *TypeScript* configuration was used or not. A sample parsed `tsconfig.json` representation is shown in Figure 6.

```
{
  "exclude": ["**/node_modules"],
  "compilerOptions": {
    "allowSyntheticDefaultImports": true,
    "baseUrl": ".",
    "declaration": true,
    "emitDecoratorMetadata": true,
    "esModuleInterop": true,
    "exactOptionalPropertyTypes": true,
    "experimentalDecorators": true,
    "forceConsistentCasingInFileNames": false,
    "incremental": true,
    "module": "commonjs",
    "moduleResolution": "node",
    "noFallthroughCasesInSwitch": false,
    "noImplicitAny": false,
    "noImplicitReturns": true,
    "noImplicitThis": true,
    "noUncheckedIndexedAccess": true,
    "outDir": "./lib",
    "removeComments": true,
    "resolveJsonModule": true,
    "skipLibCheck": true,
    "sourceMap": true,
    "strictBindCallApply": false,
    "strictFunctionTypes": true,
    "strictNullChecks": true,
    "strictPropertyInitialization": true,
    "target": "ES6",
    "types": ["jest", "node"],
  }
}
```

Figure 6. Compiler Options Sample

B5 Once the configuration file was parsed, we saved the data as part of the project's metadata document, maintaining a comprehensive record of configuration values.

The verification and parsing of `tsconfig.json` files were fully automated within the implemented data collection program. No manual intervention or post-processing was required at any stage. All error handling and fallback mechanisms were programmatically defined to ensure consistent behavior in the cases of misformed or missing configuration files.

5.3 Quality Assessment

This step of the research involves collecting quality metrics for each repository identified during the data mining stage. To evaluate the quality of the code of the projects analyzed in this study, identified code samples were submitted to *SonarQube* for automated analysis (SonarSource, 2025). This process is described step by step below. A local *SonarQube* server was used to host the metrics before they were parsed and stored in the database. The analysis was performed with the stable version 10.6 of *SonarQube*, using the Quality Profile provided for *TypeScript*, as available in the installation. This profile includes rules that detect bugs, code smells, and other bad practices related to the language. These steps are as follows:

C1 First, a *SonarQube* configuration file named (`sonar-project.properties`) was created in the root directory of each project. This file included the necessary settings for *SonarQube* to perform the quality analysis. Figure 7 presents an example of the configuration file used. The `key` field is automatically generated and contains a unique identifier for each project.

```
sonar.host.url=http://localhost:9000
sonar.token=sqa_594a95c0f2bff33913bc212a5a8763e96734352c
sonar.sourceEncoding=UTF-8
sonar.inclusions=**/*.ts,**/*.tsx
sonar.projectName=material-ui
sonar.projectKey=4014ed0b-d785-4780-860d-b6284176b889
```

Figure 7. Sonar Configuration File Sample

C2 Next, the *SonarQube* CLI tool (`sonar-scanner`) was executed to analyze the *TypeScript* code. This tool reads the configuration file and submits the project data to the *SonarQube* server for analysis. The server then processed the code and generated the corresponding quality metrics.

C3 Once the analysis was completed, the program accessed the *SonarQube* server API to retrieve the quality metrics. These metrics included information on code smells and bugs.

C4 The metrics retrieved were then parsed into a standardized format to be stored in our database. This step involved extracting relevant data points from the *SonarQube* results and converting them into an internal format.

C5 Finally, the parsed quality metrics were stored in the *MongoDB* database and assigned to their respective projects. This involved updating the project records to include the newly obtained metrics.

5.4 Statistic Analysis

This section outlines the methodology used to extract and analyze the data collected in the steps of section 5, focusing on the correlation between various *TypeScript* compiler options and the code quality metrics provided by *SonarQube*. Initially, all metrics produced by *SonarQube* in Section 4

were normalized by calculating the number of occurrences per thousand lines of code (KLOC), a widely used metric to compare quality between projects of different sizes (Jalote, 2025). This approach provided not only the absolute count of each metric but also a standardized measure of code quality between projects of varying sizes, enabling fair comparisons between projects with different amounts of source code.

Following data normalization, the distribution of each *SonarQube* metric was calculated and plotted. Repositories with zero occurrences of the metrics analyzed were intentionally treated separately and an additional metric was calculated to visualize the percentage of repositories in which the metric had a zero value. This approach provided insights into the prevalence of each issue throughout the dataset. We extracted the raw data stored in the database into a *Pandas* data frame to calculate the data distribution. Outliers were removed using the Interquartile Range (IQR) method to enhance the robustness of the analysis by excluding extreme values that could skew the results.

Next, we calculate the frequency of each compiler option in all repositories to determine their overall usage. This involved summing the occurrences of each option across all repositories. For more granular information, we segmented the data according to whether the `strict` compiler option was enabled or disabled. For each segment, we computed the usage statistics of individual strict-related options. This process included creating separate data frames for repositories with `strict` enabled and disabled, followed by calculating the frequency and percentage usage of each option related to strict within these groups. Bar charts were created to visualize the usage frequency of each compiler option, both overall and within the `strict` enabled/disabled segments. Logarithmic scaling was applied where necessary to accommodate wide variations in usage frequencies.

After analyzing both the quality metrics and the compiler option datasets individually, we performed a correlation analysis to quantify the relationship between *TypeScript* compiler options and the quality metrics obtained from *SonarQube*. The analysis focused on identifying which compiler options, as described in Section 4, were correlated with improvements or deterioration in code quality. Using *Pandas*, we computed the Pearson correlation coefficients between each *TypeScript* compiler option and the quality metrics. The *Pandas* `corr` function was used to calculate the pairwise correlation of columns in the data frame, providing a statistical measure of how two variables vary in relation to each other. The correlation results were then plotted to visually represent the strength and direction of the relationships between compiler options and code quality metrics. Figure 8 presents a simplified version of the *Python* script used for this purpose.

Although the primary objective of this study was exploratory, the correlation coefficients were computed using the Pearson method, which assumes linear relationships among variables and independence of observations. Outliers were handled using the IQR method to mitigate skewness effects. However, no formal statistical significance testing (e.g., p-values or confidence intervals) was performed, as the analysis was designed to identify broad empirical patterns rather than to conduct hypothesis testing. Future replications may incorporate formal significance testing and assumption

```python
values = [
  'typescriptOptions.allowUnreachableCode',
  'typescriptOptions.allowUnusedLabels',
  'typescriptOptions.exactOptionalPropertyTypes',
  'typescriptOptions.noFallthroughCasesInSwitch',
  'typescriptOptions.noImplicitOverride',
  'typescriptOptions.noImplicitReturns',
  'typescriptOptions.noPropertyAccessFromIndexSignature',
  'typescriptOptions.noUncheckedIndexedAccess',
  'typescriptOptions.noUnusedLocals',
  'typescriptOptions.noUnusedParameters',
  'typescriptOptions.noImplicitAny',
  'typescriptOptions.noImplicitThis',
  'typescriptOptions.strict',
  'typescriptOptions.alwaysStrict',
  'typescriptOptions.strictNullChecks',
  'typescriptOptions.strictBindCallApply',
  'typescriptOptions.strictFunctionTypes',
  'typescriptOptions.strictPropertyInitialization',
  'typescriptOptions.useUnknownInCatchVariables',
  'sonarMetrics.bugs'
]

data_frame_values = data_frame[values]

# Calculate the correlation matrix
correlation_matrix = -1 * data_frame_values.corr()

# Focus on the correlation of each option with bugs
bugs_correlation = correlation_matrix[sonarMetric].drop(sonarMetric)

# Convert correlation results to a DataFrame for easier handling
correlation_df = bugs_correlation.sort_values(ascending=False).reset_index()
correlation_df.columns = ['TypeScriptCompilerOptions', 'CorrelationWithBugs']

# Plotting the correlations
sns.barplot(
  data = correlation_df,
  x = 'CorrelationWithBugs',
  y = 'TypeScriptCompilerOptions'
)

plt.title('Correlation of TypeScript Copmiler Options with Bugs')
plt.xlabel('Correlation with Bugs')
plt.ylabel('TypeScript Compiler Options')
plt.savefig('correlation_with_bugs_plot.png')
```

Figure 8. Correlation Analysis Code Snippet

verification to complement the correlational findings with inferential statistical validation.

The statistical procedure was structured in two stages. First, descriptive exploratory analyzes were conducted to summarize the distribution of bugs, code smells, and compiler options across the collected repositories. These analyzes provided an empirical overview of the dataset and informed the interpretation of the subsequent inference step. Second, the correlation analysis examined the relationships between the compiler configuration patterns and the quality metrics observed.

5.5 Reproducibility and Data Availability

To ensure transparency and allow independent replication, all artifacts developed in this study are publicly available at `https://github.com/thomazmz/tsconfig-scanner`. The repository includes the complete source code of the automated data collection and analysis pipeline, comprising:

1. the *TypeScript/Node.js* scripts used for *GitHub* mining and compiler option extraction,
2. the *SonarQube* configuration files and CLI commands used for static quality assessment, and
3. the Python code used for statistical analysis and visualization.

Each step of the pipeline can be executed independently, allowing partial or full reproduction of the dataset and the results. Versioning information for the main tools used is

explicitly documented, ensuring that the execution environment can be reproduced. This open repository serves both as a replication package and as a foundation for future extensions of this study.

6 Results and Discussion

This section presents the results obtained after performing the research methodology described in Section 5. These results are organized to directly address the research question stated in Section 1 — *How do different TypeScript compiler options correlate with code quality?*. Consequently, the following subsections describe the characteristics of the dataset, configuration patterns, and the observed correlations between compiler options and quality metrics. Through our data mining process, we identified a total of 4,215 qualifying projects, which were analyzed in subsequent research stages. The average number of *TypeScript* lines per project was 2,531, and the average number of *GitHub* stars per project was 96.

6.1 Quality Characteristics of Projects

After using *SonarQube* for comprehensive static analysis, we gathered data on bugs and code smells and also extracted a composite metric, called *issues*, which represents the cumulative sum of bugs and smells in each project. This section presents an analysis of each of these metrics, supported by graphical representations to facilitate a clear understanding of their distribution and frequency across different projects.

Figure 9 illustrates the distribution of bugs per 1,000 lines of code (bugs/KLOC) across the repositories. This graph includes only those repositories that contain at least one bug, totaling 1,967 out of 4,215 repositories (46.6%). To generate this graph, we calculated the number of bugs per KLOC for each repository. Outliers were removed using the IQR method to ensure a more accurate representation of the data.

The resulting distribution exhibits a right-skewed pattern, indicating that most repositories have a relatively low number of bugs per 1,000 lines of code (bugs/KLOC), with a few repositories showing significantly higher bug densities. We observed a peak frequency at approximately 1 bug/KLOC, suggesting that while bugs are present, they are generally not pervasive across most projects. Repositories with higher bug densities may reflect specific circumstances, such as less rigorous testing practices, more complex codebases, or less experienced development teams.

Figure 10 illustrates the distribution of code smells per 1,000 lines of code (smells/KLOC). We plotted only the repositories that contained at least one code smell, totaling 3,740 out of 4,215 repositories (88.7%). As with the bug distribution, this graph also shows a right-skewed pattern, with most repositories clustering at the lower end of the smell-density spectrum. This suggests that although code smells are more prevalent than bugs, as evidenced by the larger number of affected repositories, their density remains relatively low in most cases. The long tail of the distribution indicates that some repositories have a significantly higher concentration of code smells, which could adversely affect maintainability and overall code quality over time.

Finally, Figure 11 presents the distribution of the composite metric: issues per 1,000 lines of code (issues/KLOC). This metric aggregates the counts of both bugs and code smells, offering a comprehensive view of the quality concerns within the projects. The graph includes only those repositories with at least one issue, comprising 3,786 of 4,215 repositories (89.7%). To generate this visualization, individual metrics were summed, normalized by lines of code, and outliers were removed using the IQR method. The resulting distribution exhibits a similar right-skewed pattern as observed in individual metrics, with the majority of repositories exhibiting a low density of issues. The distribution shown in the graph reinforces the observation that, while issues are common, their overall impact on most projects is relatively contained. However, a subset of repositories still exhibits high issue densities.

6.2 Configuration Characteristics of Projects

In this section, we present a comprehensive overview of the *TypeScript* configuration options identified in the studied projects. Specifically, we detail the compiler options employed and highlight their frequency and usage patterns. Our objective is to provide a detailed perspective on how different *TypeScript* options are adopted, providing information on current configuration trends across the selected projects. We focus on three main aspects: the overall usage of compiler options, their usage when the `strict` compiler option is disabled, and their usage when the `strict` option is enabled. Evaluating these settings individually is important, as it helps us understand the different approaches developers take toward type-checking and code quality, particularly when they explicitly choose to disable `strict` mode.

Figure 12 presents the overall usage of *TypeScript* compiler options in the 4215 repositories analyzed. To generate this graph, we extracted the compiler options stored in our database and transformed the data into a data frame, computing the total usage count for each compiler option in all repositories. The options were then sorted in descending order according to usage frequency and plotted as a bar chart. The `strict-related` compiler options are highlighted in bold to improve readability.

From this graph, we observed that certain `strict` options, such as `strictNullChecks`, `noImplicitAny`, and `noImplicitThis`, are among the most frequently used in projects. This suggests a trend towards adopting stricter type-checking measures to enhance code quality. In contrast, options such as `allowUnreachableCode` and `allowUnusedLabels` are rarely used, indicating that developers tend to prioritize configurations that enforce more rigorous coding standards.

The graph also reveals that there is limited customization in the enabled or disabled areas within the strict rule group. Most repositories tend to enable the default `strict` compiler options without making significant modifications to the individual options within the group. This may suggest that developers prefer the default configuration provided by *TypeScript* or are not fully aware of the granular control they have over

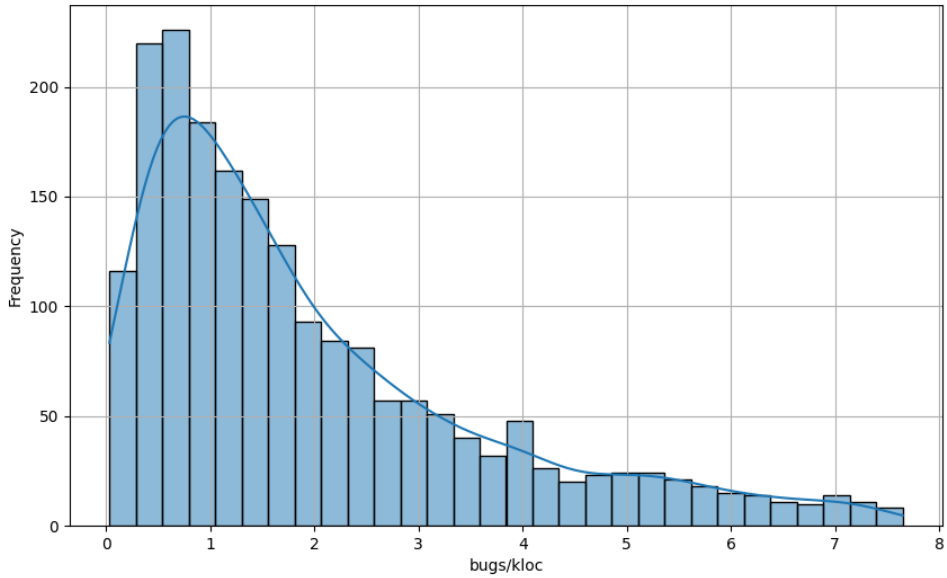

Figure 9. Bug Distribution

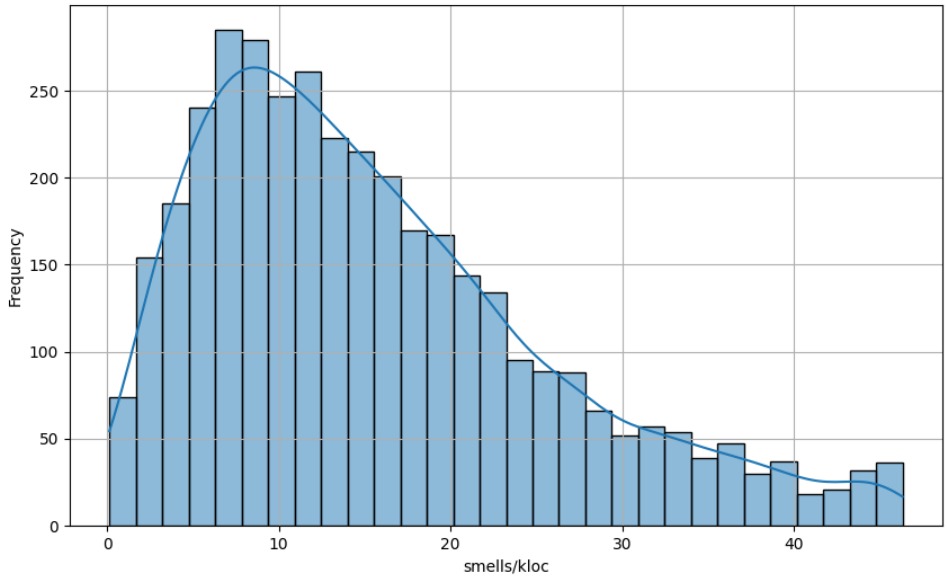

Figure 10. Smell Distribution

these settings.

Additionally, we observed that some compiler options that would increase strictness are not enabled in many projects. This contrasts with the fact that the default `strict` rules are enabled in most repositories. For example, options such as `exactOptionalPropertyTypes` and `noUncheckedIndexedAccess`, which offer additional strictness, are rarely used.

This suggests that while developers value the baseline strictness of the default settings, they may hesitate to enable finer-grained options that require substantial code changes or workflow adjustments. Another explanation is recency: several low-adoption, type-related options such as `noUncheckedIndexedAccess`, `exactOptionalPropertyTypes`, and `noPropertyAccessFromIndexSignature`, were introduced in *TypeScript* 4.x, and many teams may not yet have evaluated or adopted them.

Figure 13 focuses on the usage of `strict`-related compiler options in projects where the `strict` compiler option is disabled, accounting for 621 of 4215 repositories (14.7%). Similarly to the process used for the previous graph, the relevant data was extracted, converted into a data frame, and the usage counts of each strict option were summed.

We observed that even when the `strict` option is disabled, certain strict-related options such as `noImplicitAny` and `strictNullChecks` are still prominently used compared to others. When `strict` configuration is disabled, none of the strict-related options is enabled by default, which requires developers to make a deliberate decision to activate them.

This pattern suggests that developers may recognize the

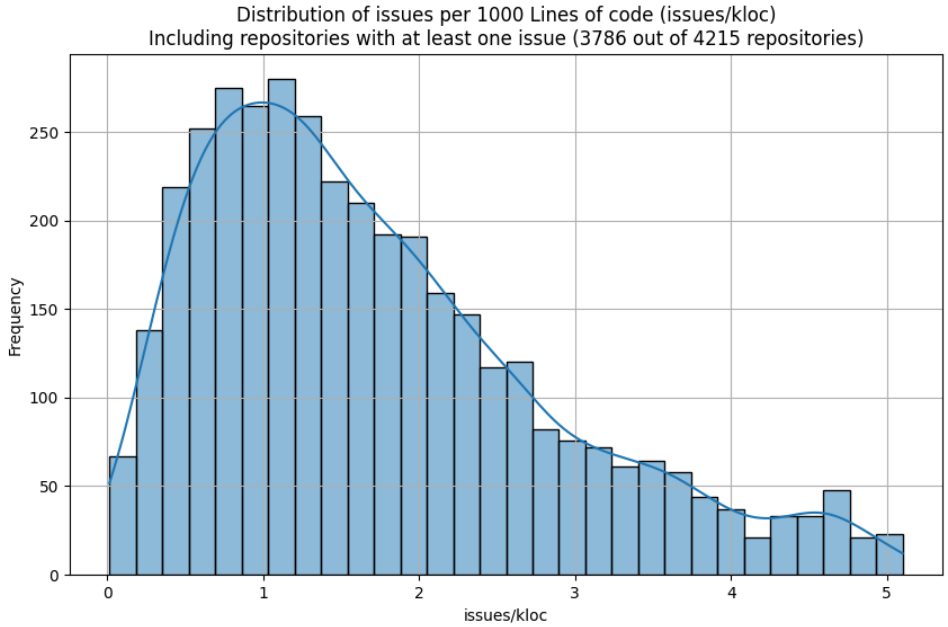

Figure 11. Issue Distribution

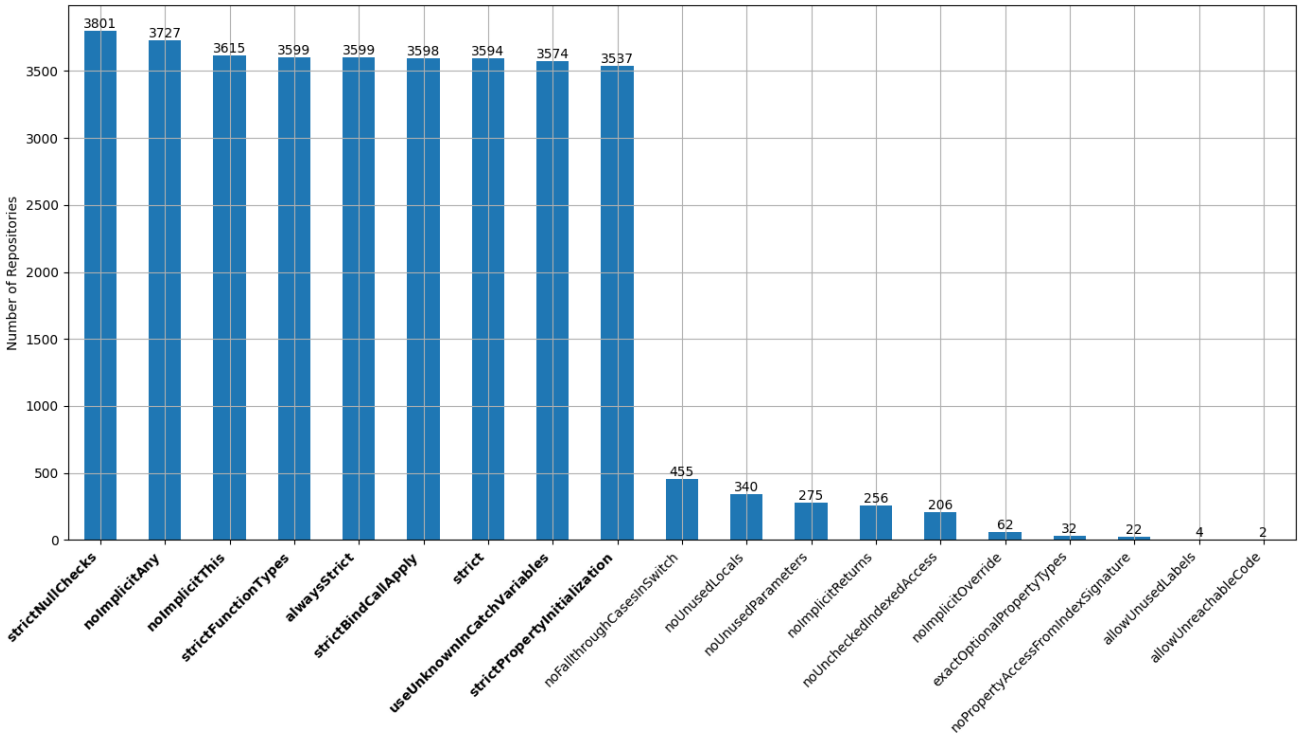

Figure 12. Use of *TypeScript* Compiler Options Across All Projects

specific benefits of these individual options, as they are often explicitly enabled. It indicates that developers prioritize the type-checking features provided by `noImplicitAny` and `strictNullChecks` to particularly enhance code robustness, even if they choose not to enforce the full set of strict compiler rules. Figure 14 examines the usage of `strict`-related compiler options in projects where the `strict` compiler option was enabled, which corresponds to a total of 3,594 of 4,215 repositories (85.2%). To better visualize differences in usage frequencies, we plotted the number of repositories using a logarithmic scale. Even projects that fully opt to use `strict` can have some strict options set to `false`. This is possible because *TypeScript* allows for granular control over individual strict options, even when the overall `strict` setting is enabled.

Developers can selectively disable specific strict rules by explicitly setting them to `false` in their configuration. This flexibility allows them to tailor the strictness of type-checking to align with the specific needs and constraints of their projects. The graph shows that when the `strict` com-

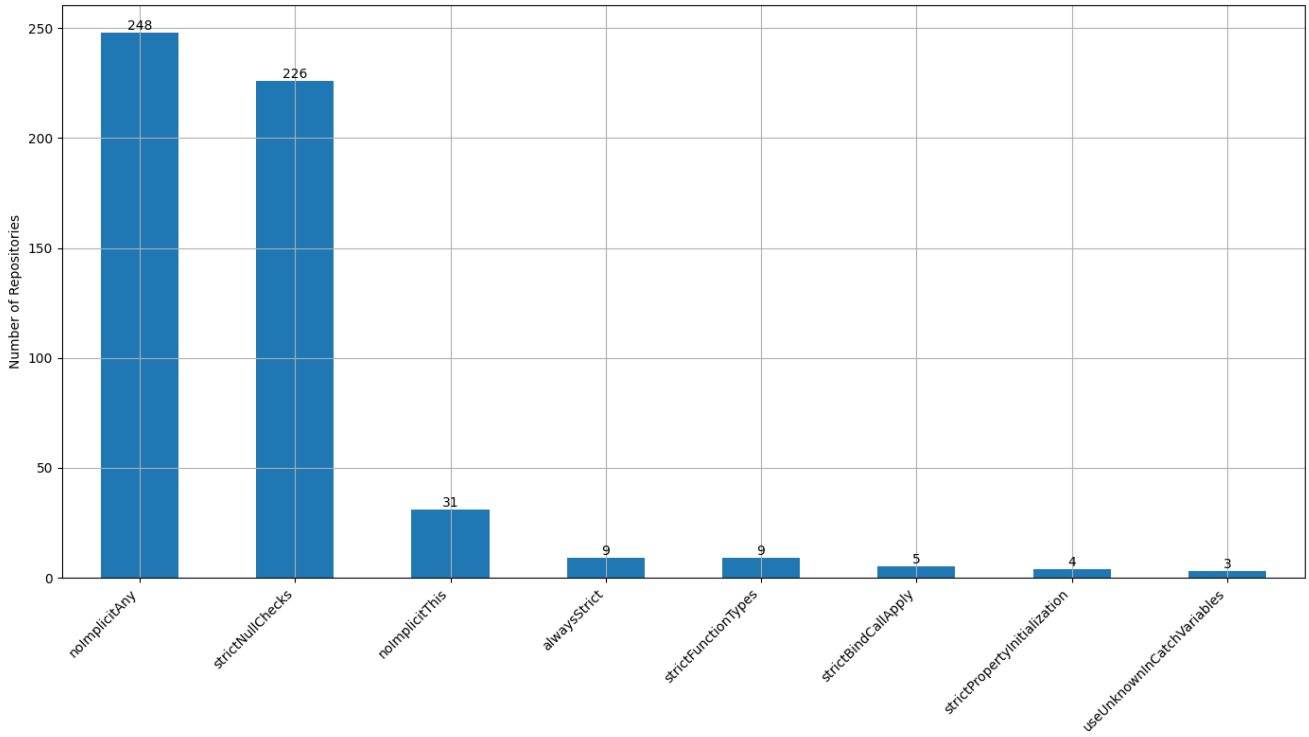

Figure 13. Use of *TypeScript* Compiler Options When `strict` is Disabled

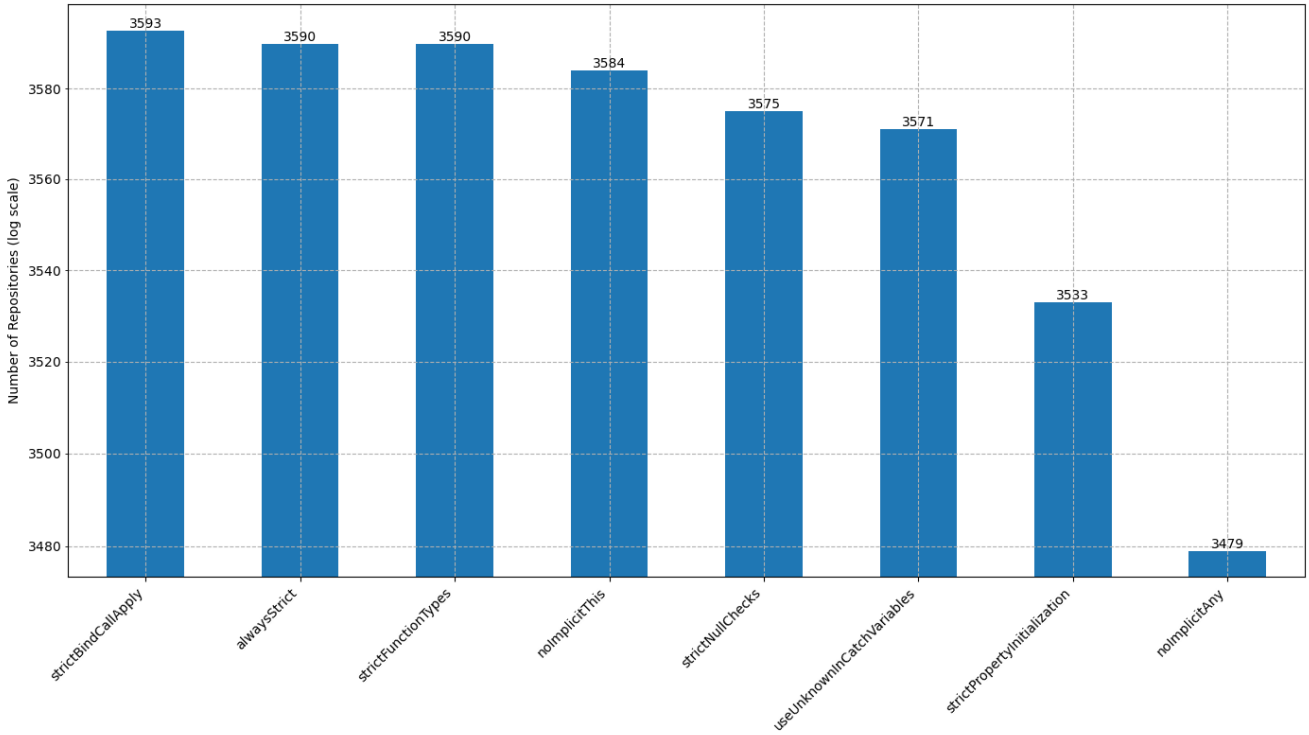

Figure 14. Use of *TypeScript* Compiler Options When `strict` is Enabled

piler option is enabled, almost all strict-related options are consistent across projects. This once again suggests that minimal customization is applied beyond the default configuration provided by the `strict` setting.

The high usage of options such as `strictNullChecks`, `strictFunctionTypes`, and `strictBindCallApply` may underscore the importance that developers place on rigorous type safety. However, it should be noted that `noImplicitAny` appears as the last bar in the third graph, indicating that it is the least frequently used `strict` option among those who enable the full set of strict compiler options. This finding is particularly interesting given that `noImplicitAny` is one of the most commonly used compiler options in general.

This contrast suggests that while developers recognize the value of `noImplicitAny` in preventing implicit `any` types, those who enable the `strict` compiler option may already

be addressing type safety through other mechanisms or prefer to maintain more granular control over this rule. It may also suggest that, within strictly typed projects, developers might opt out of enforcing `noImplicitAny` to preserve flexibility in their code or due to the presence of legacy codebases where the application of this rule universally could be impractical. This selective use of strict options highlights the nuanced decisions developers make to balance rigorous type safety with practical development considerations.

It is important to clarify the apparent inconsistency observed in Figure 14, where the `noImplicitAny` option, although it being one of the most widely used options in general, appears less common when the `strict` option is enabled. This pattern does not indicate a methodological artifact, but rather results from how the *TypeScript* compiler handles redundancy between options. When `strict` is set to `true`, `noImplicitAny` is automatically enforced as part of the strict type-checking group, making an explicit declaration redundant. Therefore, many projects omit this from their configuration files. In some cases, developers may deliberately set `noImplicitAny` to `false` even in strict mode to allow selective flexibility in legacy or hybrid codebases. Together, these two effects explain the lower explicit frequency of `noImplicitAny` observed in strictly configured repositories.

6.3 Correlation Analysis Result

This subsection provides the core quantitative results that answer the research question by examining the statistical relationships between specific *TypeScript* compiler options and the quality metrics defined in subsection 4.3.

This study presents a correlation analysis between various *TypeScript* compiler options and code quality metrics derived from *SonarQube*. Specifically, we analyze the correlation of *TypeScript* compiler options with bugs, code smells, and issues. The *TypeScript* compiler options of interest were selected as features, along with the specific *SonarQube* metrics that represent bugs and code smells. For each quality metric, we calculated a correlation matrix to identify the relationship between the *TypeScript* compiler options and the selected metric. The correlation values were then sorted and plotted to visually represent the strength and direction of these correlations.

The graph in Figure 15 displays the correlation between *TypeScript* compiler options and the number of bugs detected by *SonarQube*. From this graph, it is evident that options such as `strictPropertyInitialization`, `alwaysStrict`, and `strictFunctionTypes` exhibit the highest negative correlations with bugs. This suggests that repositories that enable these options tend to exhibit fewer bugs in the codebase. In contrast, options such as `allowUnusedLabels` and `noFallthroughCasesInSwitch` show a positive correlation, indicating that their use could be associated with an increased number of bugs.

Figure 16 illustrates the correlation between *TypeScript* compiler options and the presence of code smells. The compiler options `noImplicitAny`, `noUnusedParameters`, and `noUnusedLocals` are among the top options negatively correlated with code smells. This correlation suggests that enforcing these stricter *TypeScript* options contributes to maintaining cleaner code with fewer code smells. On the other hand, options such as `allowUnusedLabels` and `noPropertyAccessFromIndexSignature` show a positive correlation with code smells, indicating that their use could be associated with a decrease in code quality.

Figure 17 shows the correlation between *TypeScript* compiler options and the number of issues identified by *SonarQube*. The analysis reveals that compiler options such as `noImplicitAny`, `noUnusedParameters`, and `noUnusedLocals` exhibit significant negative correlations with issues, implying that enabling these options tends to reduce the number of issues within the codebase. In contrast, options such as `allowUnusedLabels` and `noPropertyAccessFromIndexSignature` display positive correlations with issues, suggesting that their usage may be associated with an increase in code-related problems.

In general, by synthesizing the results in all three proposed quality metrics, we observed that options such as `strictPropertyInitialization`, `alwaysStrict`, and `strictFunctionTypes` were strongly negatively correlated with bugs, suggesting that enabling these options helps reduce the number of bugs in the codebase. Additionally, `noImplicitAny`, `noUnusedParameters`, and `noUnusedLocals` were particularly effective in minimizing code smells and general issues, contributing to cleaner and more maintainable code. By identifying specific compiler options associated with improved code quality, these findings offer a possible explanation for the results of previous studies, such as (Bogner and Merkel, 2022), which showed that projects tend to exhibit fewer bugs and better maintainability due to the static typing features provided by *TypeScript*.

Options that allow for more flexibility, such as `allowUnusedLabels`, `noFallthroughCasesInSwitch`, and `noPropertyAccessFromIndexSignature`, were positively correlated with higher bug counts, code smells, and general problems. This highlights the potential downsides of lenient configurations that do not enforce strict type safety and coding standards. Similarly, previous studies have observed that static typing reduces the number of errors, which aligns with our findings on the benefits of strict type checking options in *TypeScript*, as reported by (Gao et al., 2017).

The results complement the findings of (Ray et al., 2017) by showing that, beyond language choice, specific compiler configurations within a language such as *TypeScript* can also have a measurable impact on code quality. Although their study emphasizes the influence of programming languages in general, our analysis refines this view by demonstrating that configuration-level decisions play a significant role in the maintainability and reliability of code. This study also reinforces the benefits of gradual typing in *TypeScript*, as discussed in previous research by (Rastogi et al., 2015), while highlighting that the enforcement of strict type-checking options leads to better results in terms of reducing bugs and code smells.

Projects adopting stricter configuration practices in the *TypeScript* compiler are typically associated with higher code quality. Projects that follow more restrictive standards tend to have fewer bugs and are more readable, making them easier to maintain over time. Avoiding overly permissive con-

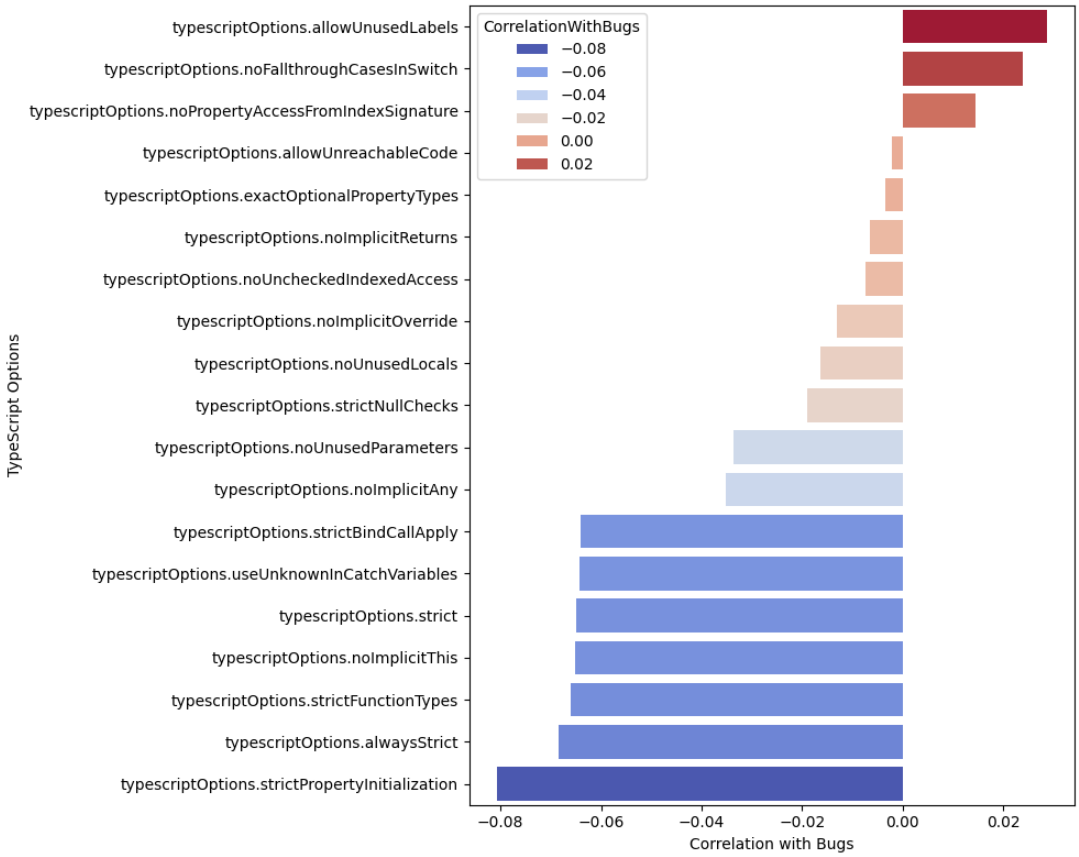

Figure 15. Correlation Between *TypeScript* Compiler Options and Bugs

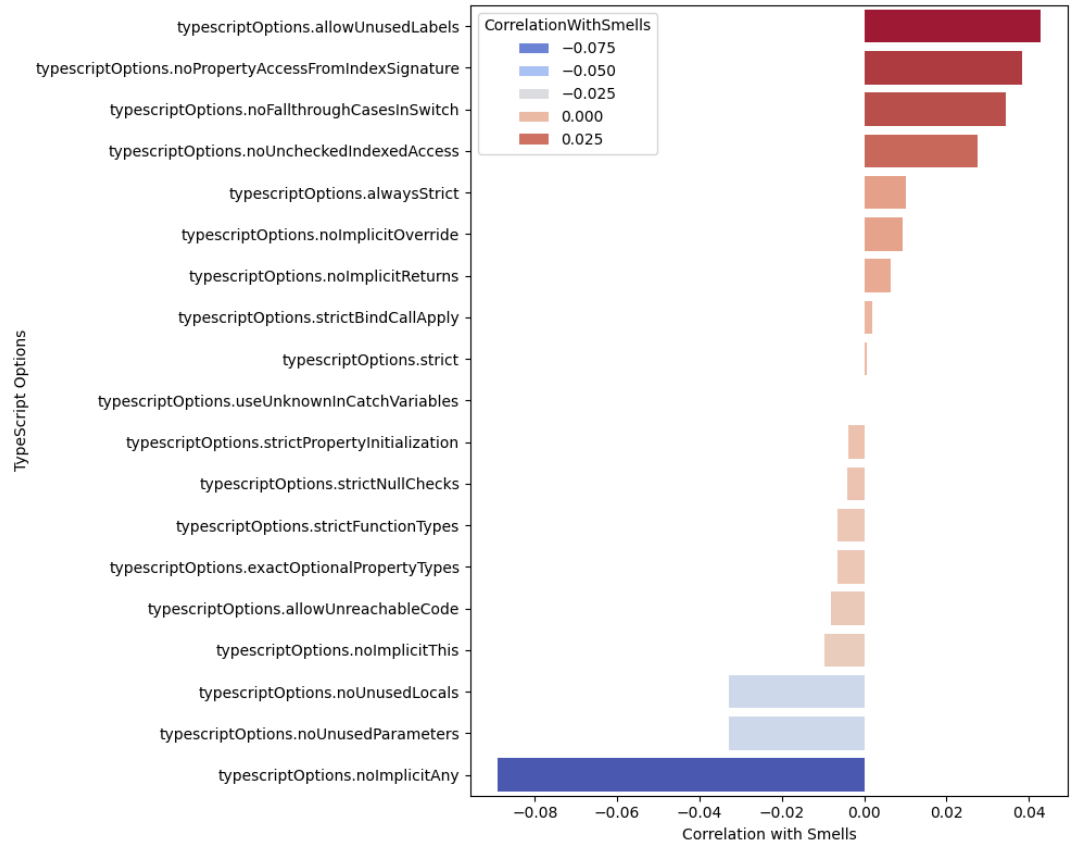

Figure 16. Correlation Between *TypeScript* Compiler Options and Smells

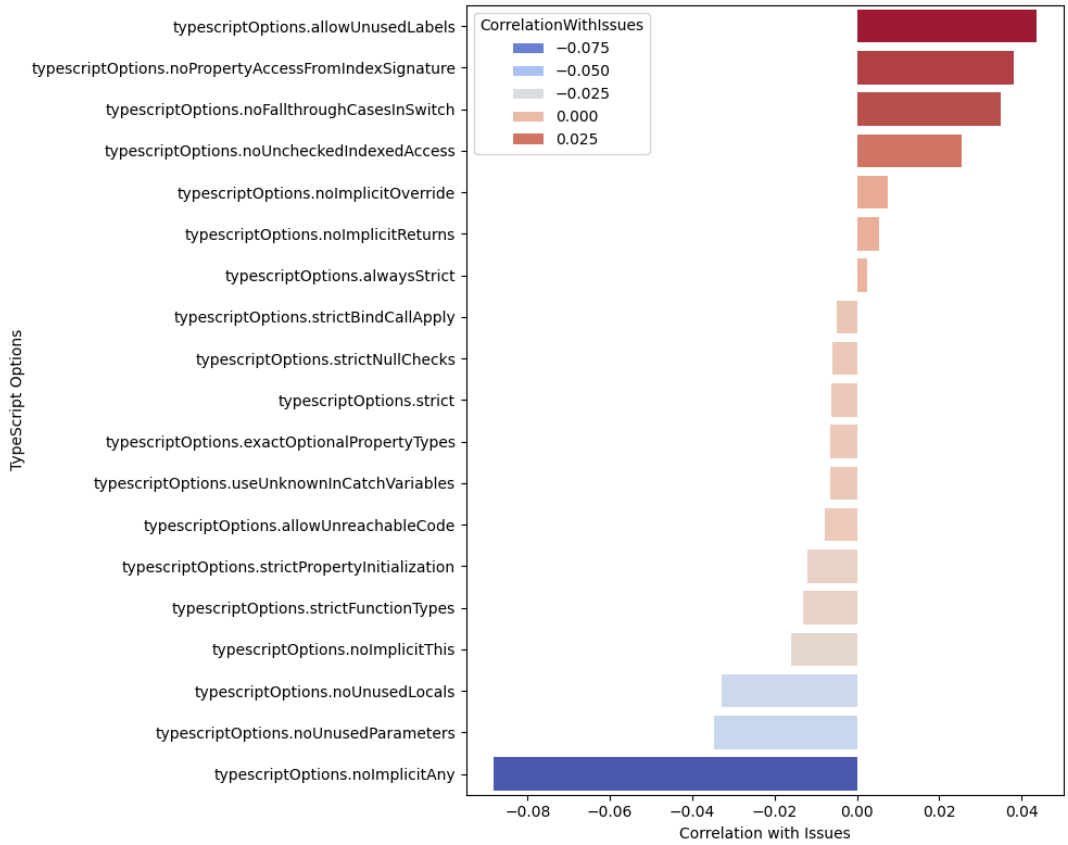

Figure 17. Correlation Between *TypeScript* Compiler Options and Issues

figurations also helps prevent common issues and enhances consistency and security in development.

Although the observed correlation coefficients are consistent across metrics, they are relatively low in magnitude, which would be expected in observational studies with multiple uncontrolled factors and high variability between projects. However, consistent patterns of negative correlation between certain options and the presence of bugs or smells provide valuable evidence of a practical association. It is important to note that this study is correlational and does not allow inference of causality. The presence of an option associated with a lower incidence of bugs does not imply that its activation alone will cause this improvement. Other contextual factors, such as team practices or project maturity, can be associated with the use of these more stringent settings.

Among the limitations of this study, we highlight the lack of control over confounding variables such as project age or size, number of collaborators, application domain, or developer experience. Such factors are known to directly affect code quality and may influence the observed results. To partially address this limitation, we plan to extend our analysis in future replications by including covariates such as project age, size (in KLOC), number of contributors, and repository popularity (e.g., *GitHub* stars and forks). These metadata fields are already available in the mined dataset and can be integrated into multivariate regression models or used for matching strategies (e.g., propensity score matching) to control for potential confounding effects. Although not applied in the present version, this enhancement will allow us to isolate the influence of compiler configuration choices from other project characteristics, thereby strengthening the robustness of our conclusions.

The following section discusses the implications of these findings for researchers and practitioners, interpreting how the observed correlations can inform compiler configuration practices, tool design, and future empirical studies.

6.4 Discussion

The correlation analysis revealed consistent empirical associations between certain *TypeScript* compiler options and static quality metrics such as bugs and code smells. Although these associations do not imply causation, they provide insight into possible configuration practices that can contribute to higher-quality code.

From a practical point of view, the findings suggest that projects that emphasize stricter compiler configurations, particularly those that enforce explicit typing and null safety, tend to exhibit fewer static issues (according to Figures 15, 16 and 17). This observation is consistent with previous research on the benefits of static typing and gradual type systems (Bogner and Merkel, 2022; Gao et al., 2017; Rastogi et al., 2015).

From a theoretical perspective, the results highlight the role of configuration-level decisions in shaping software quality, extending previous discussions that focused primarily on language-level characteristics (Ray et al., 2017). The partial use of strict options (e.g., enabling *noImplicitAny* even when *strict* is disabled) suggests a pragmatic balance adopted by developers between safety and flexibility (accord-

ing to Figures 13 and 14).

For researchers, these findings open avenues to explore developer motivations for configuration choices, tool-assisted recommendations (e.g., static analysis tools that can automatically suggest optimal compiler configurations), and the longitudinal evolution of configuration practices. For practitioners, the results reinforce the importance of informed configuration management as part of the software quality assurance process.

Finally, although this study relied on correlation analysis rather than causal inference, emerging patterns offer hypotheses for future confirmatory research, such as controlled experiments or regression-based studies incorporating project-level covariates.

6.5 Practical Contributions and Impact

Beyond the empirical correlations presented in this study, our findings offer actionable insights for professionals configuring *TypeScript* projects. By observing consistent negative correlations between specific compiler options and static issues, we identify a set of frequently effective configurations that can serve as an empirical baseline for teams aiming to improve code quality and maintainability:

- `strictPropertyInitialization`: Set it to `true` to ensure that all class properties are initialized before use.
- `exactOptionalPropertyTypes`: Set it to `true` to treat optional properties as distinct from `undefined`, requiring explicit inclusion when needed.
- `allowUnreachableCode`: Set it to `false` to flag and eliminate dead code.
- `strictFunctionTypes`: Set it to `true` to enforce stricter checking of function type variance.
- `strictNullChecks`: Set it to `true` to require explicit handling of *null* and *undefined*.
- `noImplicitThis`: Set it to `true` to require explicit, well-typed `this` context.
- `noImplicitAny`: Set it to `true` to prevent implicit `any` by requiring explicit types.
- `noUnusedParameters`: Set it to `true` to prevent unused declaration of parameters.
- `noUnusedLocals`: Set it to `true` to prevent unused declaration of variables.

This configuration is derived from the compiler options most strongly associated with reduced bugs, code smells, and general issues in our dataset. It combines settings that empirically correlate with higher code quality, emphasizing type safety, code cleanliness, and runtime robustness. These observations are correlational and do not establish causation, but provide an evidence-based starting point for practitioners.

Adopting this baseline does not imply a one-size-fits-all solution. Each project should adapt configurations according to its size, domain, and maturity. However, empirical evidence from over 4,000 real-world repositories suggests that enabling these options is consistently associated with more reliable, maintainable, and safer *TypeScript* codebases. Practitioners can use this configuration as a reference for new projects or as a guide when refactoring existing ones, aligning compiler settings with established software quality principles.

7 Threats to Validity

As with any large-scale empirical study, this research is subject to several potential threats to validity.

Internal validity. The analysis depends on the accuracy of the data extracted from *GitHub* and *SonarQube*. Although automation minimized human error, misclassifications or incomplete metadata could still occur. Furthermore, by excluding repositories that extend `tsconfig` files, we may have introduced a bias toward simpler or less modular projects. Resolving extended configuration chains, particularly when they reference external packages, would require recursive downloads and reprocessing of configurations across projects, leading to an entirely new dataset. This limitation may affect the representativeness of the analyzed configurations.

Construct validity. The study relies on *SonarQube* metrics (bugs and code smells) as proxies for code quality. The free/-community edition used in this research does not differentiate issue severity levels, which restricts our ability to compute weighted or severity-aware indicators. Future replications could address this limitation by adopting *SonarQube* editions that expose detailed severity data (e.g., critical, major, minor).

External validity. The findings are based exclusively on public *GitHub* repositories that satisfy specific inclusion criteria (e.g., having a root-level `tsconfig.json`). These conditions improve consistency, but may limit generalization to private or industrial projects, which could have different coding standards and configuration practices. Moreover, *TypeScript* is an evolving language, and future compiler versions may introduce new options that change the observed correlations.

Validity of the conclusions. The reported correlations are statistically significant but do not imply causation. Other contextual variables, such as project maturity, team size, or domain, were not controlled and can influence code quality. Thus, results should be interpreted as empirical associations rather than causal relationships.

Despite these limitations, the methodological rigor adopted through automation, reproducibility, and large coverage of the dataset helps mitigate these threats and strengthens the reliability of the findings. Future replications will incorporate control variables such as project age, size, contributor count, and domain to mitigate confounding effects on observed correlations.

8 Conclusion

This study investigated the correlations between various *TypeScript* compiler options and *SonarQube*-derived code quality metrics, such as bugs and code smells. The analysis aimed to identify which compiler options contribute to improvements or deterioration in repository code quality. Based

on the research question *How do different TypeScript compiler options correlate with code quality?*, the study revealed significant associations between specific compiler settings and quality outcomes. These findings underscore the critical role of carefully selecting *TypeScript* compiler options to enhance the overall quality of the code.

We acknowledge that there is considerable potential for further research to explore how the adoption of compiler options evolves over time and their long-term impact on large codebases. Additionally, since *TypeScript* is a language that continuously evolves, it represents a moving target. As a result, the related findings require an ongoing reevaluation to remain in line with the latest developments in the language and its surrounding ecosystem. Moreover, the introduction of new compiler options in future *TypeScript* releases underscores the need for continued analysis and adaptation.

For future research, it would be beneficial to replicate this analysis using severity-aware *SonarQube* outcomes (critical/major/minor). This would allow for the exploration of weighted variants of Sonar Findings. Furthermore, to ensure reproducibility and facilitate the evolution of this study, it is recommended that complete tables and the corresponding scripts be included in the replication package.

In addition to the correlations found, the results of this study reinforce the need to raise developers' awareness about *TypeScript* compilation options and their impact on code quality. It was noted that many projects use default configurations or strict mode without more refined customizations, which may suggest both a lack of awareness of the tuning possibilities and a choice for simplicity. However, less popular configurations have shown the potential to further improve code quality when applied correctly. This scenario points to an opportunity to encourage more careful configurations, contributing to more solid, secure, and sustainable development environments.

This research provides a scientific basis for understanding how compiler settings influence code quality in *TypeScript* projects, filling a gap in the existing literature. The methodology proposed in this study and the findings could be applied to the development of compiler settings recommendation tools, assisting development teams in building more robust and maintainable software.

Acknowledgements

The authors would like to thank the Coordination for the Improvement of Higher Education Personnel (CAPES), Brazil, and the National Council for Scientific and Technological Development (CNPq), Brazil, for their support of this work.

References

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., and Penix, J. (2008). Using static analysis to find bugs. *IEEE software*, 25(5):22–29.

Bierman, G., Abadi, M., and Torgersen, M. (2014). Understanding typescript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27.

Black, N. (2020). Boris cherny on typescript. *IEEE Software*, 37(2):98–100. Accessed: 2025-04-14.

Bogner, J. and Merkel, M. (2022). To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 658–669, New York, NY, USA. Association for Computing Machinery.

Bonino Quintana, J. and Fagerlind, S. (2020). Datamining github: Examining time-to-solve for issues in relation to group-size. KTH DivA-Portal.

Chacon, S. and Straub, B. (2014). *Pro Git*. Apress, 2nd edition.

Chess, B. and McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79.

Desmazières, A., Di Cosmo, R., and Lorentz, V. (2025). 50 years of programming language evolution through the software heritage looking glass. In *Mining Software Repositories 2025*.

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE.

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., and Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126:106347.

Gao, Z., Bird, C., and Barr, E. T. (2017). To type or not to type: Quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, ICSE '17, pages 758–769. IEEE, IEEE Press.

GitHub, Inc. (2025). Github docs. https://docs.github.com/. Accessed: 2025-04-14.

Goldberg, J. (2022). *Learning TypeScript*. O'Reilly Media.

Hu, H., Wang, Y., Rubin, J., and Pradel, M. (2025). An empirical study of suppressed static analysis warnings. *Proceedings of the ACM on Software Engineering*, 2(FSE):290–311.

International, E. (2025). Ecmascript language specification. https://tc39.es/ecma262/. ECMA-262, 12th Edition, 2025.

International Organization for Standardization and International Electrotechnical Commission (2024). ISO/IEC 25002:2024 systems and software engineering — systems and software quality requirements and evaluation (square) — quality model overview and usage. https://www.iso.org/standard/78175.html. International Standard, Edition 1, March 2024.

Jalote, P. (2025). Industry-strength software. In *A Concise Introduction to Software Engineering: With Open Source and GenAI*, pages 1–22. Springer.

JetBrains (2024). The state of developer ecosystem in 2024. https://www.jetbrains.com/lp/devecosystem-2024/. Accessed: 2025-04-14.

New, M. S., Licata, D. R., and Ahmed, A. (2021). Gradual type theory. *Journal of Functional Programming*, 31:e21.

Oliveira, A., Oizumi, W., Sousa, L., Assunção, W. K., Garcia, A., Lucena, C., and Cedrim, D. (2022). Smell patterns

as indicators of design degradation: Do developers agree? In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, pages 311–320.

Prescott, K. (2018). How we gradually migrated to typescript at unsplash. https://medium.com/unsplash/how-we-gradually-migrated-to-typescript-at-unsplash-7a34caa24ef1. Accessed: 2025-04-14.

Rastogi, A., Swamy, N., Fournet, C., Bierman, G., and Vekris, P. (2015). Safe & efficient gradual typing for typescript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 167–180, New York, NY, USA. Association for Computing Machinery.

Ray, B., Posnett, D., Devanbu, P., and Filkov, V. (2017). A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100.

SonarSource (2025). Sonarqube documentation. https://docs.sonarsource.com/sonarqube/latest/. Accessed: 2025-04-14.

Stack Overflow (2024). 2024 stack overflow developer survey: Programming, scripting, and markup languages. https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages. Accessed: 2025-04-14.

Tilkov, S. and Vinoski, S. (2010). Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83.

TypeScript Project (2025). Typescript docs - documentation. https://www.typescriptlang.org/docs/. Accessed: 2025-04-14.

W3Techs (2025). Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/. Accessed: 2025-04-14.

Wang, Z., Fang, Y., and Wang, N. (2025). An empirical study on bugs in typescript programming language. *Journal of Systems and Software*, 226:112445.

Wirfs-Brock, A. and Eich, B. (2020). Javascript: The first 20 years. In *Proceedings of the ACM on Programming Languages (HOPL-IV)*. ACM.