On the Impact of Typing on Code Smelliness: An Empirical Comparison Between JavaScript and TypeScript Projects

David Leclerc
Polytechnique Montréal
Montréal, Canada
david.leclerc@polymtl.ca

***Abstract*—Code smells are a recurrent problem in the software engineering industry. TypeScript introduces a strict type system to the JavaScript context, which promises a reduction in fault-proneness of apps. This study explores the differences in smell-proneness and evolution across the 25 most recent patch versions of 30 JavaScript and 30 TypeScript open-source projects. We find that the use of TypeScript leads to a reduction in the number of smells found in JavaScript-based apps. We also observe that this reduction is durable as applications gain in size. However, TypeScript does not seem to stimulate code smell removal over time in comparison with JavaScript.**

***Index Terms*—code smells, JavaScript, TypeScript, software quality, strict type systems, comparative study, empirical study, exploration study**

I. Introduction

Due to the explosion of the Internet starting in the 90's, JavaScript has gained a lot of traction within the software engineering community over the course of the past two decades. Initially meant as a way to add basic interactive elements on a web page, it has evolved a great deal and is now powering all kinds of very popular technologies and frameworks, some of which are used by the most successful organizations in the world. So much so that, for the 9th year in a row, JavaScript has been deemed the most commonly used programming language, according to the 2021 *Developer Survey* of Stack Overflow. [10] Nevertheless, the language has its downsides: its dynamic and typeless nature make it easy for developers to introduce code smells (i.e. poor choices in terms of software design) into their codebases. This is why various individuals and organizations have started to come up with strictly typed supersets of JavaScript, such as Microsoft with their now increasingly adopted TypeScript language. They claim those extensions to help developers write more readable and maintainable code. [11]

As prior research has shown, bugs are often linked with poor design choices when it comes to software development, a.k.a. code smells. [6], [4] If the code written in the context of strictly typed programming results in better code, we wonder if the use of TypeScript translates in a significant impact on the presence and evolution of code smells within JavaScript-based applications.

To the extent of our knowledge, no prior research has addressed the impact of using TypeScript on the presence of code smells within JavaScript-like apps. The objective of this article is to fill this gap in the literature by empirically exploring this question from two angles: that of the smell-proneness of, and the evolution of smells in such apps, in a comparative perspective. More specifically, our study addresses two research questions:

(RQ1) Does TypeScript reduce the smell-proneness of JavaScript-based applications?

(RQ2) Does TypeScript have an impact on the evolution of smells in JavaScript-based applications?

The remainder of this paper is divided as follows: section II lists the various types of code smells that we detect. In section III, we give details about our subject systems, as well as specify the different steps we take to extract the data required for our analyses. The results we obtain are presented in section IV. Section V discusses the different types of validity threats that apply in the context of our study. Previous work is discussed in section VI and conclusions are given in the final section VII.

II. Background

In order to study the impact of strict typing on the quality of open-source JavaScript/TypeScript software, we first identify a list of smells that can be detected in both languages alike. We begin our search based off the work of Johannes *et al.*, who report a set of 12 widespread JavaScript code smells. [3] Due to limitations in terms of detection abilities, we only keep 7 of those smells, namely: *Lengthy Line*, *Long Parameter List*, *Depth*, *Long Method*, *Assignment in Conditional Statement*, *Complex Switch Case*, and *Complex Code*. We extend our list with 18 new smells, which we not only deem highly problematic, but also easily preventable. The latter is especially true thanks to modern IDEs, which can provide intelligent feedback given a solid type declaration system, as is the case for TypeScript, thanks to static code analysis. In this section, we give a brief description for each of the smells we consider during our study. For more details about said smells, including the Sonar rules used to track them, we kindly refer the reader to table VIII in the appendix.

Lengthy Line When a code line exceeds 180 characters, it is deemed too long. This bad practice, referred to as the *Lengthy*

Line smell, decreases the code's readability, thus making it harder to understand.

Long Parameter List When a function has too many input parameters, it is either taking on too many responsibilities, or the app lacks an appropriate data structure. This smell is detected when a given function expects more than 7 parameters.

Depth When the nesting level of control flow statements exceeds 3, the code becomes increasingly hard to comprehend. Such bad design can easily lead to the commonly known code smell *Spaghetti Code*.

Long Method A method whose size exceeds 200 lines of code is considered too long. This smell is symptomatic of a function which takes on too many responsibilities.

Assignment in Conditional Statement Making assignments in sub-expressions is generally regarded as a bad practice, as they make the code less readable. Ideally, such expressions should not have side-effects. Moreover, assignments in conditional statement can lead to the accidental creation of global variables.

Complex Switch Case When `switch` statements are composed of many cases, they become hard to understand. Such statements are usually synonymous with entangled data mappings, and should be split. This smell is detected when `switch` statements have more than 30 non-empty cases.

Complex Code The complexity of functions should be kept to a minimum, for maintainability purposes. When the cyclomatic complexity metric exceeds 10 for a given function, it is deemed too complex, resulting in the detection of the *Complex Code* smell.

Lengthy File Bloated files are hard to maintain and understand. When a given file has more than 1,000 lines of code, it is considered too big, and should be split into multiple smaller files.

Magic Number When an arbitrary number is used somewhere in the code without any explanations, said number is called magic. This is usually considered bad practice, as it hinders the code's maintainability. Developers should store arbitrary numbers in constants with relevant names, so that other contributors can better understand their meaning and reuse them elsewhere in the code.

Retired Code Having commented out code bloats up files and reduced readability. It also creates confusion as to why said code has not simply been removed. Such code should be deleted from codebases.

Complex Expression It is considered a bad coding practice when an expression includes more than 3 binary (i.e. `&&` or `||`) or ternary operators (i.e. `X ? Y : Z`). It creates confusion among developers, given that it makes the code more complex, and thus less readable.

Shadowed Variable When a variable declared in an outer scope is overridden, or shadowed, it creates confusion and hinders readability. It allows for potential bugs, as developers who interact with this part of the code may think they are dealing with a single variable, when it isn't the case.

Empty Function A function whose body is empty is smelly. If a comment specifies why that is, then it isn't considered smelly. However, in all other cases, such functions may lead to unexpected behavior (i.e. bugs) if used in production.

Duplicated String It is considered a bad practice to have multiple Duplicated String in the code, as it forces the developers to update all instances at once. This smell is detected when more than 3 instances of a given string exist in the code. Similarly to the scenario described by the *magic number* smell, it is better to declare meaningful constants and use them instead of duplicating strings.

Weak Equality Strong comparison operators (i.e. === and `!==`) should always be used, given that JavaScript performs type coercion before comparing values. For instance, the expression `' ' == 0` is evaluated as true. This behavior can sometimes be useful, yet it more often leads to confusion and bugs.

Unreachable Code Code that is out of reach due to jump statements is smelly. It bloats the method where it exists, while being useless. It hinders readability, and can lead developers to think that the operations it described are actually carried on, and thus introduce bugs.

Useless Assignment The scenario where a variable is assigned a value only to be overwritten shortly thereafter is referred to as a useless assignment. If the variable is never accessed, said assignment is also deemed useless. Such behavior is smelly, because it is either pointless, or symptomatic of an involuntary coding error.

Overwritten Built-Ins Overwriting standard, built-in objects is most likely to have catastrophic effects on previously working code, and should be avoided at all cost. This smell is detected when any of the native JS/TS objects (e.g. `Object`, `Date`, `Promise`, etc.) is overridden.

Overwritten Variable/Function Re-declaring an already existing variable/function is a bad practice, because it makes the code confusing. Developers may think they are using the variable/function that was first declared, when they actually are accessing another one. This smell is referred to as *overwritten variable/function*.

Ordinal String Comparison The use of ordinal comparison operators (i.e. <, <=, >=, >) on strings is prone to yielding unexpected results, and should therefore be avoided. For instance, the expression `'10' < '2'` will be evaluated to true by JavaScript. This is because the default behavior of JavaScript when it comes to compare strings is to use alphabetical ordering.

Invariant Function A function which has many `return` statements, yet always outputs the same value, is smelly. In the best case, such behaviour is symptomatic of poor design. In the worst case, it is a development error.

Invalid Error Throwing any object that is not derived from the `Error` class is bad practice. The `Error` class should always be used, as it has been specifically designed to provide developers with enough details about unexpected behavior, especially stack traces.

Unknown Output When a function does not return anything, its output should not be used in the code. Doing otherwise is guaranteed to lead to unexpected behavior, and should be avoided.

Inconsistent Return The use of `return` statements should be consistent throughout a function, that is whatever the path taken by the control flow. If this isn't the case, then different types may be returned by a single function, which can be very confusing to the user, and make said function hard to maintain.

Duplicated Function It is considered bad practice to have several functions implementing the exact same functionality. When functions are duplicated and a change is required, all instances need to be updated at the same time. If the developer in charge of refactoring forgets one of the instances, a bug is introduced.

III. Study Design

The *goal* of our study is to investigate and compare the occurrence of code smells in JavaScript and TypeScript projects. The *quality focus* is the fault-proneness of the source code, which can have a negative effect on the cost of maintenance and evolution of the system if high. The *perspective* is that of researchers, interested in the impact of strict typing on the software quality of systems built using the aforementioned languages. The results of this study are also of interest for developers, who need to forecast and optimize the time and efforts required to uphold and expand such systems. Testers may also find an interest in our study, as previous literature has shown how code smells can be correlated with fault-prone code. Given that code smells have often been found to be correlated with fault-prone code, managers and quality assurance teams should finally find a particularly great interest in knowing whether using TypeScript can improve the quality, and thus reduce the cost-of-ownership of the systems for which they are responsible. The *context* of this study consists of 25 types of code smells, identified in the 25 patch versions of 30 JavaScript and 30 TypeScript projects. In the following, we introduce our research questions, describe the studied systems, and present our data extraction approach. We then proceed to describe our model construction and analysis approaches.

A. Studied Systems

In order to answer our research questions, we perform a case study on a set of 30 JavaScript and 30 TypeScript projects of open-source nature. The entirety of these projects can be accessed via their public repositories on GitHub. We select the most popular projects based on their stargazers count, and only consider those for which there exists at least 25 patch versions. Due to restrictions imposed by the API of our smell detector, i.e. SonarQube, on the maximum number of issues that can be retrieved for a given project, we discard any repository for which more than 10,000 issues are detected in any of their 25 most recent patch versions. As a result, we have to reject a total of 44 JavaScript and 62 TypeScript projects. We also set aside any project that is either a style guide, or consists in a dictionary of good and bad practices (of which there are a few), in order for our sample to be representative of applications that are being developed in a real-world context. For more details about our subject systems, we refer the reader to tables IX and X in appendix, which respectively summarize the characteristics of the selected JavaScript and TypeScript projects.

B. Data Extraction

As a means to determine TypeScript's impact on the quality of software, we consider the distribution of code smells in our subject systems. We do so for the 25 most recent patch versions of said systems at the file level. Due to the lack of readily available software dedicated to the detection and extraction of code smells in the version history of JavaScript applications, we combine a small number of tools and build a pipeline that allows for the extraction of that data. We describe in further details how we use each of those tools in the following sub-sections. For the interested reader, we include two diagrams describing our pipeline in appendix (see figures 2 and 3). We also make said pipeline public on Github[1].

Repository Mining We rely on GitPython and GitHub's REST API to mine the repositories of our subject systems. We use the former to clone repositories and checkout specific project versions, based on the associated commit hashes. We use the latter to extract the list of version tags of all projects, which we then proceed to manually filter. More specifically, we solely keep the tags corresponding to the 25 most recent patch versions of each project.

Smell Detection We choose to work with SonarQube when it comes to detecting code smells in our subject systems. Each smell is defined by a unique Sonar rule violation, as reported in table VIII. Rule violations are either boolean- or threshold-based. We use SonarQube's default set of thresholds, as defined in its factory configuration, and report the associated metric values in section II.

We scan every project for issues (i.e. smells) using a local instance of a SonarQube server, and fetch the results upon completion using the associated web API. Once this is done, we discard some of the detected smells based on two criteria. First, we only keep smells that were found in files whose extension matches the language of the project to which they belong. For the considered mapping of file extensions, please see table I. Second, we ignore all smells found in test files. We identify such files based on whether or not the word 'test' can be found anywhere in their absolute path.

TABLE I
File Extensions Mapping

Language	File Extensions
JavaScript	.js, .jsx, .mjs
TypeScript	.ts, .tsx

[1]*https://www.github.com/dleclercpro/LOG6306*

IV. Analysis and Results

(RQ1) Does TypeScript reduce the smell-proneness of JavaScript-based applications?
We divide our first main research question into three sub-questions, so as to consider the differences in smelliness between JavaScript and TypeScript projects from various angles.

(RQ1.1) Are TypeScript apps and files less likely to be affected by code smells than JavaScript ones?
To answer this sub-question, we first consider the fraction of projects affected by code smells in both languages. We find a total of 1 and 3 projects that are entirely free of smells for JavaScript and TypeScript, respectively.

We then compare the distribution of code smells within TypeScript and JavaScript projects from two viewpoints: the ratio of apps and that of versioned file instances (e.g. two versions of a single file count as two different instances) which are affected by every smell type. We report those distributions in tables II and III respectively.

We find a reduction in the occurrence of every smell type in TypeScript projects from both viewpoints. The only exception to this statement has to do with the smell *Ordinal String Comparison*, which is found in 2 TypeScript projects, while it is nowhere to be seen in the set of JavaScript projects.

The greatest drop *wrt.* occurrence on the scale of apps is observed for the smells *Assignment in Conditional Statement* and *Duplicated String*, which respectively affect 40% and 33.3% less TypeScript apps than JavaScript ones. In terms of versioned file instances, we find a reduction in the occurrence of smells *Magic Number* and *Complex Code* of 13.7%, 8.2%.

Those first results suggest that TypeScript apps and files are in fact less likely to be affected by any of the smells considered in this study than apps and files written in plain JavaScript.

One possible general explanation as to why some smells (i.e. *Magic Number*, *Assignment in Conditional Statement*, and *Duplicated String*) are less likely to appear in TypeScript apps and files could be that developers who consciously choose to work with type systems when they don't have to (i.e. they could simply use JavaScript) are more sensitive to the importance of following good coding practices, and therefore inherently better programmers. In other cases, e.g. *Magic Number*, it may be that a type system helps developers in conceiving well-adapted data structures, which consequently simplifies the code they have to write in order to manipulate said structures. Obviously, those attempts at explaining the aforementioned results are only hypotheses that warrant further exploration and studies.

(RQ1.2) Does TypeScript reduce the likelihood of code smell co-occurrences in comparison with JavaScript?
To further explore the potential impact of type systems on code smelliness, we follow the work of Peruma *et al.* and investigate the two-way co-occurrence of smell types in our subject systems. [5] In order to compute those, we use the implementation of the *apriori* algorithm from the `mlxtend` Python library.

TABLE II
Volume of Apps Exhibiting Each Smell in JavaScript and TypeScript Projects

Smell	**Apps (JS)**	**Apps (TS)**
Lengthy Line	76.7%	63.3%
Long Parameter List	23.3%	20.0%
Depth	86.7%	66.7%
Long Method	70.0%	46.7%
Assignment in Cond. Statement	73.3%	33.3%
Complex Switch Case	6.7%	-
Complex Code	93.3%	76.7%
Lengthy File	40.0%	26.7%
Magic Number	93.3%	86.7%
Retired Code	70.0%	56.7%
Complex Expression	80.0%	53.3%
Shadowed Variable	93.3%	70.0%
Empty Function	66.7%	56.7%
Duplicated String	86.7%	53.3%
Weak Equality	63.3%	36.7%
Unreachable Code	6.7%	3.3%
Useless Assignment	56.7%	30.0%
Overwritten Built-Ins	50.0%	30.0%
Overwritten Variable/Function	23.3%	20.0%
Ordinal String Comparison	-	6.7%
Invariant Function	16.7%	13.3%
Invalid Error	20.0%	16.7%
Unknown Output	3.3%	3.3%
Inconsistent Return	93.3%	86.7%
Duplicated Function	56.7%	50.0%

We find that most code smell co-occurrences are rather low (see tables XI and XII in appendix). In the context of JavaScript, we find the highest co-occurrence to be 17.1% for the pair of smells *Complex Code* and *Magic Number*. In that of TypeScript, that same pair of smells also has the highest co-occurrence value, at 7.8%. We report the top-3 co-occurrences found in both JavaScript and TypeScript applications alike in table IV. We observe a reduction of all top-3 co-occurrences in the context of TypeScript applications, when compared to JavaScript ones. However, because the absolute value of said co-occurrences is very low, this reduction may be the result of a statistical artefact: the fact that there are simply less code smells in TypeScript than in JavaScript projects is bound to lead to lower co-occurrences.

Summary for RQ1: It appears as though TypeScript does indeed reduce the smell-proneness of JavaScript-based applications. Our evidence suggests that using TypeScript results in a drop of all smell types, both at the level of apps and individual files, with the small exception of *Ordinal String Comparison*, which was found in 2 of our TypeScript subject systems, yet none of the JavaScript ones. The smells who

TABLE III
VOLUME OF FILES EXHIBITING EACH SMELL IN JAVASCRIPT AND TYPESCRIPT PROJECTS

Smell	Files (JS)	Files (TS)
Lengthy Line	2.3%	1.5%
Long Parameter List	0.4%	0.2%
Depth	4.8%	1.3%
Long Method	2.5%	0.5%
Assignment in Cond. Statement	2.1%	0.7%
Complex Switch Case	-	-
Complex Code	11.0%	2.8%
Lengthy File	0.4%	0.2%
Magic Number	22.8%	9.1%
Retired Code	2.8%	0.8%
Complex Expression	3.4%	1.2%
Shadowed Variable	8.1%	2.5%
Empty Function	2.0%	1.2%
Duplicated String	6.3%	1.4%
Weak Equality	2.4%	0.7%
Unreachable Code	0.1%	-
Useless Assignment	1.9%	0.3%
Overwritten Built-Ins	1.7%	0.3%
Overwritten Variable/Function	0.3%	0.1%
Ordinal String Comparison	-	-
Invariant Function	0.1%	0.1%
Invalid Error	0.4%	0.3%
Unknown Output	-	-
Inconsistent Return	8.1%	2.8%
Duplicated Function	1.1%	0.6%

TABLE IV
TOP-3 CO-OCCURRING SMELLS IN JAVASCRIPT/TYPESCRIPT PROJECTS

Smells	Probability (JS)	Probability (TS)
(CC, MN)	17.1%	7.8%
(MN, SV)	11.8%	4.2%
(MN, IR)	11.1%	4.9%

accuse the biggest reduction in occurrence are *Assignment in Conditional Statement* and *Duplicated String* on the app scale, and *Magic Number* and *Complex Code* on that of versioned file instances.

(RQ2) Does TypeScript have an impact on the evolution of smells in JavaScript-based applications? We investigate this research question from two viewpoints. First, we look at the smell count differences between consecutive pairs of patch versions. We do so for the 25 most recent versions of our subject systems on both the app and the file scales. Second, we compare the spread of code smells within an application as its size increases, depending on whether it has been written using JavaScript or TypeScript. We use various general linear models to try and estimate the impact of the total number of lines of code, language used, as well as interaction factors between these two to explain the total number of detected smells in a project.

(RQ2.1) Does TypeScript have an impact on the smell trends across patch versions?
To study the spread of code smells throughout a project's evolution, we use a variant of the approach taken by Peruma *et al.* in their 2019 paper. [5] More specifically, we only consider JavaScript and TypeScript smelly files which exist over the entirety of the 25 most recent patch versions of each project. For each commit involving a given file, we compute whether there is a difference in its code smell count compared to the previous commit. Depending on the result, we assign one of the following tags to the current commit: 'steady', 'increased', or 'decreased'. The number of times each of those tags occurs is then counted on two scales: for a single app, as well as for a single file. The obtained distributions are shown in tables V and VI.

TABLE V
SMELL TRENDS EXPRESSED AS NUMBER OF CONSECUTIVE APP VERSIONS (OUT OF 24)

Metric	Min.	Q1	Median	Mean	Q3	Max.
Trends in Smelly JavaScript Apps (29/30 Projects)						
Steady	2.0	9.0	12.0	12.3	15.0	23.0
Increased	1.0	5.0	8.0	8.4	11.0	17.0
Decreased	0.0	2.0	3.0	3.3	4.0	9.0
Trends in Smelly TypeScript Apps (27/30 Projects)						
Steady	10.0	14.5	17.0	17.0	20.5	23.0
Increased	1.0	3.0	5.0	5.4	7.5	10.0
Decreased	0.0	0.0	1.0	1.6	2.5	9.0

TABLE VI
SMELL TRENDS EXPRESSED AS NUMBER OF CONSECUTIVE FILE VERSIONS (OUT OF 24)

Metric	Min.	Q1	Median	Mean	Q3	Max.
Trends in Smelly JavaScript Files (1,043/80,821 Files)						
Steady	9.0	23.0	24.0	23.2	24.0	24.0
Increased	0.0	0.0	0.0	0.5	1.0	11.0
Decreased	0.0	0.0	0.0	0.3	0.0	7.0
Trends in Smelly TypeScript Files (441/96,286 Files)						
Steady	13.0	23.0	24.0	23.5	24.0	24.0
Increased	0.0	0.0	0.0	0.3	0.0	9.0
Decreased	0.0	0.0	0.0	0.2	0.0	3.0

We observe that the smell count shows more variability in the context of JavaScript applications than in that of TypeScript. On average, JavaScript applications will stay steady during 12.3 commits, whereas TypeScript ones will do so over 17.0. They also tend to increase and decrease more than the smell counts reported for TypeScript projects. On average, the number of smells detected in JavaScript applications will increase and decrease over 8.4 and 3.3 commits respectively. On the other hand, that same number of smells in TypeScript applications will increase and decrease over 5.4 and 1.6 commits respectively.

The same does not remain true when it comes to smell trends between consecutive file versions. We observe that both JavaScript and TypeScript files stay steady over the entirety of the considered scope, with TypeScript ones being slightly less prone to variability. In other words, once a smell is introduced to a file, it persists: either it has been written in JavaScript or TypeScript does not seem to matter.

It appears that TypeScript apps are less prone to variability than JavaScript ones. Moreover, using a static type system in the context of JavaScript-based programming does not seem to encourage the removal of code smells by development teams.

(RQ2.2) Does TypeScript slow down the spread of code smells in a project as it grows in comparison to JavaScript? In order to further explore the differences between TypeScript and JavaScript projects in terms of smell-proneness, we compute three different linear regression models. The first model (1) aims at explaining the total number of smells in an app (S) as a function of the line of codes it contains (LOC), where:

$$S \ = \ b_{LOC} \cdot LOC + K$$

where K is a general constant, the intercept of the linear model 1. The total of $N = 1,500$ apps – that is, 30 projects in each language, with 25 versions per project (the most recent ones) – are considered in this analysis. This first model yields a good fit to the data $(F(1, 1, 498) = 1992.134(p < 0.001), R^2 = 0.571)$. As shown in table VII, the data confirms a positive and statistically significant relationship between the total number of lines of codes in an app and the total number of smells it contains $(b_{LOC} = 0.012(p < 0.001))$.

We then explore whether, in addition to the sheer number of lines of codes an app contained, the mere fact that it is written in TypeScript as opposed to JavaScript can make a difference. For this purpose, we estimate a second linear regression model (2) where:

$$S \ = \ b_{LOC} \cdot LOC + b_{TS} \cdot TS + K$$

where TS is a dummy variable that takes the value 1 if the app is written in TypeScript, and 0 if written in JavaScript. This model yields a significantly better fit to the data in comparison with model 1 $(F_{change}(1, 1, 497) = 110.497(p < 0.001), R^2 = 0.600)$. As shown in table VII, the data still confirms a positive and statistically significant relationship between the number of lines of codes in an app and the total number of smells it contains $(b_{LOC} = 0.012(p < 0.001))$. In addition, this model suggests a net reduction in the total number of smells in an app if said app is written in TypeScript as opposed to JavaScript $(b_{TS} = -127.458(p < 0.001))$. In other words, smell-proneness is significantly lessened in TypeScript programs than it is in JavaScript programs, even after controlling for the total lines of code a program contains.

Finally, we want to explore a potential interaction effect between the number of lines of codes a program contains, and the language it is written in. If significant, this interaction effect would mean that the number of smells would increase at a different pace according to the the number of lines of codes in programs written in JavaScript as opposed to TypeScript. For this purpose, we estimate a third linear regression model (3) where:

$$S \ = \ b_{LOC} \cdot LOC + b_{TS} \cdot TS + b_{TSxLOC} \cdot TSxLOC + K$$

where $TSxLOC$ is an interaction term that is the multiplication of the dummy variable TS and the number of lines of codes contained in an app. Again, this model yields a significantly better fit to the data in comparison with model 2 $(F_{change}(1, 1, 496) = 145.198(p < 0.001), R^2 = 0.636)$. As shown yet again in table VII, the data reconfirms a positive and statistically significant relationship between the number of lines of code in an app and the total number of smells it contains $(b_{LOC} = 0.014(p < 0.001))$ as well as a net, albeit somewhat weaker, reduction in the total number of smells in an app if said app is written in TypeScript as opposed to JavaScript $(b_{TS} = -38.472(p < 0.01))$. In addition, the interaction term between the total number of lines of code and the language in which they are coded proves a significant predictor of the number of smells in an app $(b_{TSxLOC} = -0.007(p < 0.001))$.

TABLE VII
MAIN AND INTERACTION EFFECTS OF APP SIZE AND PROGRAMMING LANGUAGE ON THE SMELL COUNT OF AN APP, CONSIDERING THE 25 MOST RECENT PATCH VERSIONS OF ALL PROJECTS ($N = 1,500$)

Model	**Predictor Variable**	***b* (Std. Error)**	***p*-value**
1	Lines of Code (LOC)	0.012 (.000)	< 0.001
	Intercept (K)	37.730 (7.367)	< 0.001
2	Lines of Code (LOC)	0.012 (.000)	< 0.001
	TypeScript (TS)	-127.458 (12.127)	< 0.001
	Intercept (K)	106.728 (9.679)	< 0.001
3	Lines of Code (LOC)	0.014 (.000)	< 0.001
	TypeScript (TS)	-38.472 (13.736)	< 0.01
	Interaction Term (TSxLOC)	-0.007 (.001)	< 0.001
	Intercept (K)	73.092 (9.656)	< 0.001

R^2: Model 1: 0.571, Model 2: 0.600, Model 3: 0.636

Dependent Variable: Smell Count in App

When considered together, this suggests that: (1) no matter what, the total number of lines of code contained in an app significantly increase the number of smells it is likely to contain; that (2) after statistically controlling for the number of lines of code in an app, programs written in TypeScript are likely to contain significantly less smells than JavaScript programs; and that (3) the number of smells in JavaScript programs increases significantly more rapidly according to increasing number of lines of code than programs written in TypeScript. This suggests that TypeScript programs are given a head start in terms of reduced smell-proneness, and that this advantage increases as programs get lengthier in terms of the lines of code they contain. This can be seen very well in figure 1.

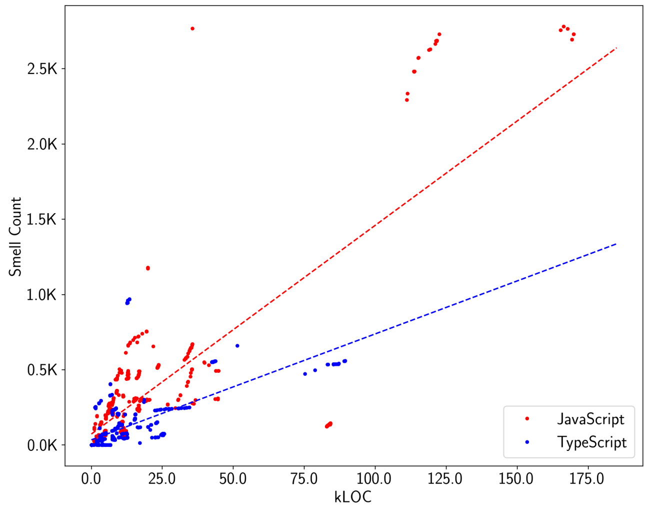

Fig. 1. Evolution of smell count in the 25 most recent releases of the 60 considered JavaScript and TypeScript projects as a function of their size, expressed in lines of code (LOC).

Summary for RQ2: It appears that TypeScript seems to have an impact on the evolution of smells in JavaScript-based apps. Not only does its use diminish the variability of the smell count in an application, it also slows down the spread of smells as said application grows. Moreover, we observe that TypeScript doesn't seem to affect the likelihood that a smell will be removed once it is introduced to a file.

V. Validity Threats

Construct Validity Given that our set of smells can be detected based off specific and objectively measurable code metrics, we would normally not consider construct validity to be a threat in the context of our study. However, the fact that SonarSource does not provide any information regarding the performance of SonarQube is a potential drawback. That being said, given that SonarQube's use is rapidly increasing across the private industry [8], we assume its performance to be well-suited for our research purposes. In any cases, SonarSource's software was the only viable option we found for smell detection purposes in the context of JavaScript-based development: the tools made available by Fard *et al.* and Johannes *et al.* could not be used, the former being no longer maintained, and the latter being unusable.

Internal Validity This validity threat refers to the correctness of the analyses we performed on the collected data. In our case, this threat has a potential effect on the conclusions we can draw about the smell-proneness of JavaScript-based applications. Indeed, due to limitations of SonarQube's web API (i.e. a maximum of 10,000 issues can be read from the API for a single project analysis), we had to discard a total of 44 and 62 JavaScript and TypeScript projects respectively, thus resulting in a potential selection bias. The reason behind this decision making is that SonarQube was never intended to be used in the way we have, and was therefore somewhat ill-suited for our research purposes.

External Validity This type of validity threat has to do with the extent to which our results can be generalized. As we mentioned before, the entirety of our subject systems comes from publicly available GitHub repositories. Each and every one of them was selected based on its stargazers count, which can be seen as a proxy for community appreciation and popularity. Therefore, we estimate that coding practices may be very different in the context of privately owned and developed applications, thus limiting the generalizability of our results beyond popular open-source projects.

Conclusion Validity Conclusion validity refers to the overall credibility of the conclusions we draw based on our analyses. In our case, the biggest caveats to the correctness of our conclusions can be found in the construct and external validity threats that we previously described. These both warrant further investigation with regards to the impact of using type system extensions, such as TypeScript, in the context of JavaScript-driven development.

VI. Related Work

Despite the fact that JavaScript is currently one of the most used programming languages in the world [10], the quantity of tools dedicated to the detection of code smells in the context of JavaScript-based development is still scarce. There exists a few static code analysis solutions on the market, yet all of them either exist as extensions of popular modern IDEs, or explicitly meant to be used within continuous integration pipelines. One of those is SonarQube, which we opted for the detection of code smells in our subject systems.

Nonetheless, a few authors have worked on the elaboration and implementation of standalone tools aimed at the detection of smells in JavaScript codebases. In 2013, Fard *et al.* introduced JSNose, which could be used by developers to spot a set of 13 metric-based JavaScript code smells. [1] In 2017, Saboury *et al.* proposed a metric-based framework which could detect a set of 12 widespread code smells, which are considered to make the code less readable. [1] They examined the effect of code smells on the fault-proneness of JavaScript server-side projects using survival analysis, and found that non-smelly files had hazard rates 65% lower than their smelly counterparts. They also found that the smells *Variable Re-Assign* and *Assignment in Conditional Statements* had the highest hazard rates. [6] In 2019, Johannes *et al.* worked on improving the resolution of the survival analyses done in Saboury *et al.*'s study by introducing a so-called 'line grain analysis'. This novel technique, which consists in intersecting code smells and faulty lines, allowed them to bring the granularity of their analyses from the file level down to that of the code line. Their results were consistent with those of their predecessors: they found that non-smelly JavaScript files had hazard rates at least 33% lower than that of their smelly counterparts. They also found the smells *Variable Re-Assign*, *Assignment in Conditional Statements*, and *Complex Code* to be the most hazardous.

To the best of our knowledge, no work has been done towards understanding the impact of strictly typed JavaScript extensions (e.g. TypeScript) on the presence of code smells. In 2017, Gao *et al.* did, however, evaluate the benefits of using static type systems on JavaScript codebases. They considered a corpus of 400 public bugs in a total of 398 JavaScript systems that were available on GitHub, and assessed to which extent bugs could have been prevented, thanks to the information provided by two static type systems, i.e. Flow 0.30 and TypeScript 2.0. They found that, on average, 15% of them could have been avoided. [2]

VII. Conclusion

The goal of this paper was to investigate the impact of strictly typed extensions of JavaScript, most specifically that of TypeScript. We did so by comparing the prevalence of 25 code smells as well as their evolution in a corpus of 30 JavaScript and 30 TypeScript projects of open-source nature over their 25 most recent patch versions. Considering the fact that code smells are found to be correlated with buggy code [6], [4], our findings are in line with those of Gao *et al.*, who provided evidence that the use of type systems can effectively help reduce the fault-proneness of apps. We found that TypeScript effectively reduces the occurrence of code smells in individual apps, as well as individual patch versions of files. Even though the co-occurrences of code smells in JavaScript applications aren't strong, we also observed that TypeScript diminishes them as well. When it comes to the evolution of smells within JavaScript-based applications, we found that TypeScript lessens the variability of the smell count in an app. We found, however, that it does not seem to help in the removal of smells from one patch version of an app to another. Finally, we designed three general linear models to predict the spread of code smells based on the size of apps, and the language used. We observed three things. First, the size of an app is directly linked to the number of smells detected for both JavaScript and TypeScript. Second, TypepScript programs contain significantly less smells than JavaScript ones. Finally, the number of code smells in a TypeScript application increases significantly slower as a function of said application's size than it does in the case of JavaScript ones.

In the future, we recommend working on the implementation of smell detection tools that are well-suited to the context of JavaScript-based applications, and can also be used outside of IDEs and CI pipelines. Considering smells and bugs tend to increase as a a project does in size, we think further research on the impact of type systems on JavaScript development is of crucial relevance for managers and developers alike, as both benefit from a reduced technical debt. To that extent, we think replicating our work with a larger sample size would prove very valuable. Addressing the issue of the maximum number of issues allowed in a single project version would help in finding more compatible subject systems. Another research avenue can be found in the replication of our work in the context of the private industry. Finally, we recommend further investigation *wrt.* the impact of TypeScript on the change-proneness of smelly code. It would be interesting to consider the differences in smell counts between consecutive versions of a file, only when said file is effectively modified, consistently with what Peruma *et al.* did in their 2019 paper. [5]

References

[1] Fard, A. M., Mesbah, A., *JSNOSE: Detecting JavaScript Code Smells*, 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2013, pp. 116-125, DOI: 10.1109/SCAM.2013.6648192.

[2] Gao, Z., Bird, C., Barr, E. T., *To Type or Not to Type: Quantifying Detectable Bugs in JavaScript*, ICSE '17: Proceedings of the 39th International Conference on Software Engineering, IEEE, 2017, pp. 758–769, DOI: 10.1109/ICSE.2017.75.

[3] Johannes, D., Khomh, F., Antoniol G., *A large-scale empirical study of code smells in JavaScript projects*, Software Quality Journal, Vol. 27, Springer, 2019, pp. 1271–1314, DOI: 10.1007/s11219-019-09442-9.

[4] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., *On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation*, Empirical Software Engineering, Vol. 23, 2018, pp. 1188–1221, DOI: 10.1007/s10664-017-9535-z.

[5] Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., Palomba, F., *On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study*, in Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, 2019, pp. 193–202.

[6] Saboury, A., Musavi, P., Khomh, F., Antoniol, G., *An empirical study of code smells in JavaScript projects*, 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 294-305, DOI: 10.1109/SANER.2017.7884630.

[7] SonarQube, *About*, accessed on December 27, 2021, URL: https://www.sonarqube.org/about/.

[8] SonarSource, *List of Customers*, accessed on December 24, 2021, URL: https://www.sonarsource.com/customers/.

[9] SonarSource, *Rules*, accessed on December 15, 2021, URL: https://rules.sonarsource.com/.

[10] Stack Overflow, *2021 Developer Survey*, accessed on December 20, 2021, URL: https://insights.stackoverflow.com/survey/2021.

[11] Wagner, Bill, *TypeScript: Enhance Your JavaScript Investment with TypeScript*, Microsoft, Vol. 29 Number 6, June 2014, accessed on November 29, 2021, URL: https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/june/typescript-enhance-your-javascript-investment-with-typescript.

TABLE VIII
LIST OF SMELLS CONSIDERED

Smell	Abbreviation	Sonar Rule	New[1]
Lengthy Line	LL	S103	No
Long Parameter List	LPL	S107	No
Depth	D	S134	No
Long Method	LM	S138	No
Assignment in Conditional Statement	ACS	S1121	No
Complex Switch Case	CSC	S1479	No
Complex Code	CC	S1541	No
Lengthy File	LF	S104	Yes
Magic Number	MN	S109	Yes
Retired Code	RC	S125	Yes
Complex Expression	CE	S1067	Yes
Shadowed Variable	SV	S1117	Yes
Empty Function	EF	S1186	Yes
Duplicated String	DS	S1192	Yes
Weak Equality	WE	S1440	Yes
Unreachable Code	UC	S1763	Yes
Useless Assignment	UA	S1854	Yes
Overwritten Built-Ins	OBI	S2424	Yes
Overwritten Variable/Function	OVF	S2814	Yes
Ordinal String Comparison	OSC	S3003	Yes
Invariant Function	IF	S3516	Yes
Invalid Error	IE	S3696	Yes
Unknown Output	UO	S3699	Yes
Inconsistent Return	IR	S3801	Yes
Duplicated Function	DF	S4144	Yes

[1]New smells are smells which had not been described in Johannes *et al.*'s paper. [3]

TABLE IX
LIST OF SELECTED JAVASCRIPT PROJECTS

Project	**Year**	**Stars**	**Forks**	**Contributors**	**Commits**	**Tags**	**JS Ratio**	**TS Ratio**
d3/d3	2010	99.6k	22.9k	121	4.3k	284	100.0%	-
facebook/create-react-app	2016	92.4k	23.4k	465	2.7k	406	98.3%	0.1%
axios/axios	2014	89.8k	9.0k	275	1.1k	53	93.4%	3.8%
typicode/json-server	2013	58.4k	5.6k	64	0.8k	125	95.9%	-
chartjs/Chart.js	2013	55.6k	11.3k	380	4.1k	88	98.4%	1.3%
jquery/jquery	2009	55.6k	20.2k	284	6.5k	153	93.6%	-
expressjs/express	2009	55.3k	9.3k	250	5.6k	280	99.9%	-
tailwindlabs/tailwindcss	2017	50.6k	2.4k	189	4.4k	188	77.4%	-
serverless/serverless	2015	41.5k	5.0k	359	14.7k	317	97.0%	0.2%
yarnpkg/yarn	2016	40.3k	2.8k	453	2.3k	170	98.7%	-
nuxt/nuxt.js	2016	39.0k	3.2k	302	5.6k	156	95.1%	0.2%
mermaid-js/mermaid	2014	39.0k	2.9k	266	4.2k	95	69.2%	-
Dogfalo/materialize	2014	38.6k	4.9k	247	3.9k	44	71.9%	-
nwjs/nw.js	2012	38.6k	3.9k	103	4.6k	304	42.8%	-
iamkun/dayjs	2018	37.4k	1.8k	268	1.4k	106	100.0%	-
GitSquared/edex-ui	2017	34.8k	1.9k	55	1.4k	31	87.8%	-
hexojs/hexo	2012	33.9k	4.2k	153	3.5k	144	100.0%	-
alvarotrigo/fullPage.js	2013	33.0k	7.4k	127	1.9k	76	93.2%	-
koajs/koa	2013	32.1k	3.1k	201	1.1k	95	100.0%	-
quilljs/quill	2012	31.6k	2.7k	122	5.2k	140	95.2%	-
zenorocha/clipboard.js	2015	31.5k	4.0k	54	0.4k	39	99.1%	0.6%
lerna/lerna	2015	31.1k	2.0k	252	2.2k	180	99.8%	-
preactjs/preact	2015	30.5k	1.7k	272	4.9k	184	95.9%	2.6%
nolimits4web/swiper	2012	29.5k	9.6k	204	2.9k	231	77.8%	10.2%
carbon-app/carbon	2017	29.3k	1.6k	157	1.7k	210	99.9%	-
webtorrent/webtorrent	2013	25.5k	2.5k	149	3.0k	377	99.7%	-
request/request	2011	25.4k	3.1k	281	2.3k	144	99.0%	-
ramda/ramda	2013	21.5k	1.4k	274	2.9k	52	99.8%	-
bower/bower	2012	15.1k	2.0k	206	2.8k	113	100.0%	-
riot/riot	2013	14.6k	1.0k	173	3.8k	253	98.6%	0.4%

TABLE X
LIST OF SELECTED TYPESCRIPT PROJECTS

Project	**Year**	**Stars**	**Forks**	**Contributors**	**Commits**	**Tags**	**JS Ratio**	**TS Ratio**
socketio/socket.io	2010	54.9k	9.9k	179	1.9k	155	-	100.0%
ionic-team/ionic-framework	2013	45.8k	13.5k	405	11.3k	299	8.5%	42.8%
remix-run/react-router	2014	45.4k	8.8k	427	5.2k	173	9.8%	89.9%
vercel/hyper	2016	37.5k	3.1k	253	2.9k	108	2.3%	96.1%
nativefier/nativefier	2015	29.3k	1.9k	116	1.1k	133	9.7%	83.7%
formium/formik	2017	29.0k	2.4k	398	1.7k	225	5.4%	92.5%
facebook/docusaurus	2017	28.9k	3.9k	464	4.3k	133	9.8%	87.1%
streamich/react-use	2018	26.9k	2.1k	161	2.9k	255	0.5%	99.5%
tannerlinsley/react-query	2019	24.3k	1.3k	370	1.4k	418	0.7%	99.3%
JedWatson/react-select	2014	23.8k	3.8k	369	4.0k	140	0.9%	99.1%
akveo/ngx-admin	2016	23.3k	7.4k	31	0.5k	27	0.3%	67.2%
pmndrs/react-spring	2018	22.0k	1.0k	131	3.2k	232	2.0%	97.7%
balena-io/etcher	2015	21.9k	1.6k	64	2.9k	168	0.5%	97.7%
react-navigation/react-navigation	2017	20.7k	4.5k	117	1.7k	1689	0.6%	98.0%
railsware/upterm	2015	19.4k	0.7k	59	3.0k	260	-	96.2%
floating-ui/floating-ui	2016	19.0k	1.3k	41	1.1k	174	33.5%	65.0%
youzan/vant	2017	18.9k	9.2k	205	8.5k	467	8.5%	60.3%
marmelab/react-admin	2016	18.6k	4.1k	382	11.2k	229	5.0%	94.8%
reduxjs/reselect	2015	18.4k	0.7k	91	0.7k	35	1.8%	98.2%
palantir/blueprint	2016	18.4k	1.9k	281	2.2k	1252	2.2%	88.5%
statelyai/xstate	2015	18.3k	0.8k	252	4.4k	186	1.0%	97.4%
graphql/graphql-js	2015	18.2k	1.9k	173	3.1k	147	2.5%	97.3%
apollographql/apollo-client	2016	17.1k	2.3k	423	10.4k	1134	1.5%	98.5%
react-dnd/react-dnd	2014	16.9k	1.8k	184	2.2k	148	1.9%	96.6%
pubkey/rxdb	2016	16.7k	0.8k	130	6.3k	93	1.3%	98.5%
pmndrs/react-three-fiber	2019	16.0k	0.9k	129	1.9k	292	0.6%	99.2%
fingerprintjs/fingerprintjs	2015	15.5k	1.8k	72	0.7k	83	5.4%	92.7%
oldj/SwitchHosts	2011	15.4k	1.8k	24	1.2k	46	1.8%	53.7%
ustbhuangyi/better-scroll	2015	15.2k	2.6k	33	2.0k	84	4.7%	66.0%
flatpickr/flatpickr	2015	14.6k	1.3k	213	2.8k	147	-	92.9%

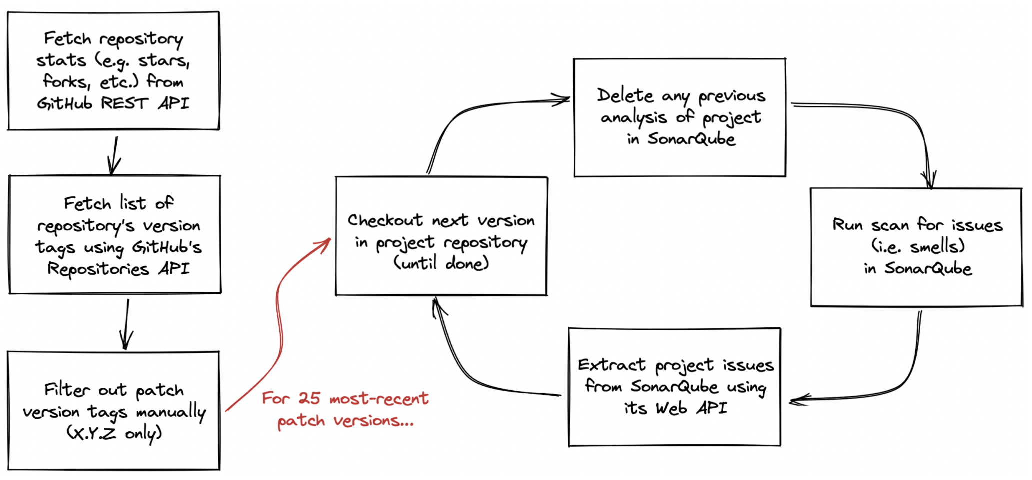
Fetch repository stats (e.g. stars, forks, etc.) from GitHub REST API
Fetch list of repository's version tags using GitHub's Repositories API
Filter out patch version tags manually (X.Y.Z only)
For 25 most-recent patch versions...
Checkout next version in project repository (until done)
Delete any previous analysis of project in SonarQube
Run scan for issues (i.e. smells) in SonarQube
Extract project issues from SonarQube using its Web API

Fig. 2. Data extraction pipeline for a single JavaScript/TypeScript project.

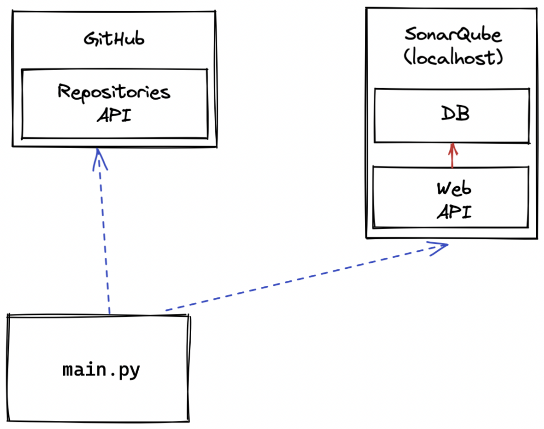

Fig. 3. Infrastructure of the data extraction pipeline.

TABLE XI
CO-OCCURRENCE MATRIX OF SMELL TYPES IN JAVASCRIPT PROJECTS

Smell	LL	LPL	D	LM	ACS	CSC	CC	LF	MN	RC	CE	SV	EF	DS	WE	UC	UA	OBI	OVF	OSC	IF	IE	UO	IR	DF
LL	-	0.1%	1.1%	1.3%	0.7%	-	2.6%	0.6%	3.4%	0.4%	0.9%	1.5%	0.2%	1.9%	0.6%	-	0.5%	0.1%	0.1%	-	-	0.2%	-	1.6%	0.6%
LPL	0.1%	-	0.4%	0.4%	0.3%	-	0.8%	-	1.0%	-	0.4%	0.6%	0.1%	0.3%	0.2%	-	0.1%	0.1%	-	-	-	-	-	0.6%	-
D	1.1%	0.4%	-	2.5%	1.5%	-	8.9%	0.6%	7.3%	1.0%	2.7%	4.2%	1.0%	2.5%	1.0%	0.1%	0.9%	0.4%	0.1%	-	0.2%	0.2%	-	2.9%	0.5%
LM	1.3%	0.4%	2.5%	-	1.0%	-	3.9%	0.6%	5.0%	1.0%	1.5%	2.0%	0.3%	2.7%	0.9%	0.1%	1.2%	0.1%	0.1%	-	-	0.2%	-	1.9%	0.8%
ACS	0.7%	0.3%	1.5%	1.0%	-	-	2.3%	0.4%	3.7%	0.4%	1.1%	2.0%	0.3%	0.7%	1.1%	-	0.6%	0.4%	0.3%	-	-	0.1%	-	2.5%	0.4%
CSC	-	-	-	-	-	-	0.1%	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CC	2.6%	0.8%	8.9%	3.9%	2.3%	0.1%	-	0.8%	17.1%	2.5%	6.0%	8.0%	1.7%	5.8%	3.1%	0.2%	1.9%	1.0%	0.3%	-	0.2%	0.4%	-	6.7%	1.1%
LF	0.6%	-	0.6%	0.6%	0.4%	-	0.8%	-	0.7%	0.2%	0.7%	0.7%	0.2%	0.6%	0.3%	-	0.3%	0.2%	0.1%	-	-	0.1%	-	0.7%	0.3%
MN	3.4%	1.0%	7.3%	5.0%	3.7%	-	17.1%	0.7%	-	4.3%	5.3%	11.8%	2.5%	9.2%	3.4%	0.2%	3.5%	1.5%	0.6%	-	0.1%	1.0%	-	11.1%	1.7%
RC	0.4%	-	1.0%	1.0%	0.4%	-	2.5%	0.2%	4.3%	-	0.5%	1.8%	0.3%	1.7%	0.9%	0.2%	1.2%	-	0.2%	-	-	0.1%	-	1.3%	0.6%
CE	0.9%	0.4%	2.7%	1.5%	1.1%	-	6.0%	0.7%	5.3%	0.5%	-	2.6%	0.5%	1.4%	0.8%	-	0.5%	0.5%	0.1%	-	-	-	-	2.4%	0.5%
SV	1.5%	0.6%	4.2%	2.0%	2.0%	-	8.0%	0.7%	11.8%	1.8%	2.6%	-	1.4%	4.0%	1.8%	-	1.4%	1.2%	0.2%	-	0.1%	0.4%	-	6.8%	1.1%
EF	0.2%	0.1%	1.0%	0.3%	0.3%	-	1.7%	0.2%	2.5%	0.3%	0.5%	1.4%	-	1.1%	0.4%	-	0.2%	0.3%	-	-	0.1%	0.1%	-	1.1%	0.3%
DS	1.9%	0.3%	2.5%	2.7%	0.7%	-	5.8%	0.6%	9.2%	1.7%	1.4%	4.0%	1.1%	-	1.3%	-	1.8%	0.3%	0.2%	-	-	0.4%	-	2.9%	1.2%
WE	0.6%	0.2%	1.0%	0.9%	1.1%	-	3.1%	0.3%	3.4%	0.9%	0.8%	1.8%	0.4%	1.3%	-	0.2%	0.9%	0.2%	0.5%	-	-	0.1%	-	2.3%	0.4%
UC	-	-	0.1%	0.1%	-	-	0.2%	-	0.2%	0.2%	-	-	-	-	0.2%	-	0.2%	-	0.1%	-	-	-	-	0.1%	0.1%
UA	0.5%	0.1%	0.9%	1.2%	0.6%	-	1.9%	0.3%	3.5%	1.2%	0.5%	1.4%	0.2%	1.8%	0.9%	0.2%	-	0.3%	0.3%	-	-	-	-	1.4%	0.8%
OBI	0.1%	0.1%	0.4%	0.1%	0.4%	-	1.0%	0.2%	1.5%	-	0.5%	1.2%	0.3%	0.3%	0.2%	-	0.3%	-	0.1%	-	-	-	-	1.8%	0.1%
OVF	0.1%	-	0.1%	0.1%	0.3%	-	0.3%	0.1%	0.6%	0.2%	0.1%	0.2%	-	0.2%	0.5%	0.1%	0.3%	0.1%	-	-	-	0.1%	-	0.4%	0.1%
OSC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IF	-	-	0.2%	-	-	-	0.2%	-	0.1%	-	-	0.1%	0.1%	-	-	-	-	-	-	-	-	-	-	-	-
IE	0.2%	-	0.2%	0.2%	0.1%	-	0.4%	0.1%	1.0%	0.1%	-	0.4%	0.1%	0.4%	0.1%	-	-	-	0.1%	-	-	-	-	0.4%	0.1%
UO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IR	1.6%	0.6%	2.9%	1.9%	2.5%	-	6.7%	0.7%	11.1%	1.3%	2.4%	6.8%	1.1%	2.9%	2.3%	0.1%	1.4%	1.8%	0.4%	-	-	0.4%	-	-	1.0%
DF	0.6%	-	0.5%	0.8%	0.4%	-	1.1%	0.3%	1.7%	0.6%	0.5%	1.1%	0.3%	1.2%	0.4%	0.1%	0.8%	0.1%	0.1%	-	-	0.1%	-	1.0%	-

Abbreviations: LL: Lengthy Line — LPL: Long Parameter List — D: Depth — LM: Long Method — ACS: Assignment in Conditional Statement — CSC: Complex Switch Case — CC: Complex Code — LF: Lengthy File — MN: Magic Number

RC: Retired Code — CE: Complex Expression — SV: Shadowed Variable — EF: Empty Function — DS: Duplicated String — WE: Weak Equality — UC: Unreachable Code — UA: Useless Assignment — OBI: Overwritten Built-Ins

OVF: Overwritten Variable/Function — OSC: Ordinal String Comparison — IF: Invariant Function — IE: Invalid Error — UO: Unknown Output — IR: Inconsistent Return — DF: Duplicated Function

TABLE XII
CO-OCCURRENCE MATRIX OF SMELL TYPES IN TYPESCRIPT PROJECTS

Smell	**LL**	**LPL**	**D**	**LM**	**ACS**	**CSC**	**CC**	**LF**	**MN**	**RC**	**CE**	**SV**	**EF**	**DS**	**WE**	**UC**	**UA**	**OBI**	**OVF**	**OSC**	**IF**	**IE**	**UO**	**IR**	**DF**
LL	-	0.1%	1.0%	0.8%	0.6%	-	1.8%	0.4%	2.3%	0.3%	0.7%	1.4%	0.2%	2.1%	0.2%	-	0.2%	0.1%	-	-	-	0.3%	-	1.0%	0.8%
LPL	0.1%	-	0.2%	-	-	-	0.5%	-	0.6%	0.2%	-	0.3%	0.1%	0.1%	-	-	-	-	-	-	-	-	-	0.4%	-
D	1.0%	0.2%	-	0.6%	1.1%	-	3.9%	0.6%	4.2%	0.6%	1.5%	2.1%	0.6%	0.8%	0.4%	-	0.1%	0.3%	-	-	-	0.2%	-	2.0%	0.2%
LM	0.8%	-	0.6%	-	0.4%	-	1.1%	0.2%	2.0%	0.3%	0.6%	0.7%	0.2%	1.5%	0.3%	-	0.2%	-	-	-	-	0.1%	-	0.4%	0.5%
ACS	0.6%	-	1.1%	0.4%	-	-	1.9%	0.3%	1.7%	0.5%	0.9%	1.3%	0.7%	0.6%	0.4%	-	0.4%	0.4%	-	-	-	0.1%	-	0.8%	0.3%
CSC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CC	1.8%	0.5%	3.9%	1.1%	1.9%	-	-	0.6%	7.8%	1.0%	3.6%	3.8%	0.8%	1.4%	0.9%	-	0.3%	0.5%	0.2%	0.1%	0.2%	0.4%	-	3.2%	0.3%
LF	0.4%	-	0.6%	0.2%	0.3%	-	0.6%	-	0.5%	0.2%	0.6%	0.5%	-	0.3%	0.2%	-	-	0.1%	-	-	-	-	-	0.3%	0.1%
MN	2.3%	0.6%	4.2%	2.0%	1.7%	-	7.8%	0.5%	-	1.9%	3.6%	4.2%	1.3%	3.0%	2.1%	0.2%	0.7%	0.5%	-	0.1%	0.1%	1.0%	-	4.9%	1.2%
RC	0.3%	0.2%	0.6%	0.3%	0.5%	-	1.0%	0.2%	1.9%	-	0.3%	0.6%	0.4%	0.4%	0.7%	-	0.4%	-	-	-	-	-	-	0.8%	0.2%
CE	0.7%	-	1.5%	0.6%	0.9%	-	3.6%	0.6%	3.6%	0.3%	-	1.3%	0.9%	0.7%	0.6%	-	0.2%	0.3%	-	0.1%	0.2%	-	-	1.6%	0.2%
SV	1.4%	0.3%	2.1%	0.7%	1.3%	-	3.8%	0.5%	4.2%	0.6%	1.3%	-	0.3%	0.8%	1.4%	-	0.1%	0.2%	0.2%	-	0.1%	0.2%	-	2.8%	0.3%
EF	0.2%	0.1%	0.6%	0.2%	0.7%	-	0.8%	-	1.3%	0.4%	0.9%	0.3%	-	0.6%	0.3%	-	0.2%	0.1%	-	-	0.2%	0.1%	-	1.2%	0.2%
DS	2.1%	0.1%	0.8%	1.5%	0.6%	-	1.4%	0.3%	3.0%	0.4%	0.7%	0.8%	0.6%	-	0.3%	-	0.3%	0.2%	-	-	-	0.2%	-	1.0%	0.9%
WE	0.2%	-	0.4%	0.3%	0.4%	-	0.9%	0.2%	2.1%	0.7%	0.6%	1.4%	0.3%	0.3%	-	-	0.2%	-	-	-	-	-	-	0.5%	0.4%
UC	-	-	-	-	-	-	-	-	0.2%	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
UA	0.2%	-	0.1%	0.2%	0.4%	-	0.3%	-	0.7%	0.4%	0.2%	0.1%	0.2%	0.3%	0.2%	-	-	0.1%	-	-	-	-	-	0.3%	0.2%
OBI	0.1%	-	0.3%	-	0.4%	-	0.5%	0.1%	0.5%	-	0.3%	0.2%	0.1%	0.2%	-	-	0.1%	-	-	-	-	-	-	0.4%	-
OVF	-	-	-	-	-	-	0.2%	-	-	-	-	0.2%	-	-	-	-	-	-	-	-	-	-	-	-	-
OSC	-	-	-	-	-	-	0.1%	-	0.1%	-	0.1%	-	-	-	-	-	-	-	-	-	-	-	-	0.1%	-
IF	-	-	-	-	-	-	0.2%	-	0.1%	-	0.2%	0.1%	0.2%	-	-	-	-	-	-	-	-	-	-	0.1%	-
IE	0.3%	-	0.2%	0.1%	0.1%	-	0.4%	-	1.0%	-	-	0.2%	0.1%	0.2%	-	-	-	-	-	-	-	-	-	0.2%	0.1%
UO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IR	1.0%	0.4%	2.0%	0.4%	0.8%	-	3.2%	0.3%	4.9%	0.8%	1.6%	2.8%	1.2%	1.0%	0.5%	-	0.3%	0.4%	-	0.1%	0.1%	0.2%	-	-	0.3%
DF	0.8%	-	0.2%	0.5%	0.3%	-	0.3%	0.1%	1.2%	0.2%	0.2%	0.3%	0.2%	0.9%	0.4%	-	0.2%	-	-	-	-	0.1%	-	0.3%	-

Abbreviations: LL: Lengthy Line — LPL: Long Parameter List — D: Depth — LM: Long Method — ACS: Assignment in Conditional Statement — CSC: Complex Switch Case — CC: Complex Code — LF: Lengthy File — MN: Magic Number
RC: Retired Code — CE: Complex Expression — SV: Shadowed Variable — EF: Empty Function — DS: Duplicated String — WE: Weak Equality — UC: Unreachable Code — UA: Useless Assignment — OBI: Overwritten Built-Ins
OVF: Overwritten Variable/Function — OSC: Ordinal String Comparison — IF: Invariant Function — IE: Invalid Error — UO: Unknown Output — IR: Inconsistent Return — DF: Duplicated Function