

Using the TypeScript compiler to fix erroneous Node.js snippets

Brittany Reid

The University of Adelaide, Australia

brittany.reid@adelaide.edu.au

Christoph Treude

The University of Melbourne, Australia

christoph.treude@unimelb.edu.au

Markus Wagner

Monash University, Australia

markus.wagner@monash.edu

Abstract—Most online code snippets do not run. This means that developers looking to reuse code from online sources must manually find and fix errors. We present an approach for automatically evaluating and correcting errors in Node.js code snippets: Node Code Correction (NCC). NCC leverages the ability of the TypeScript compiler to generate errors and inform code corrections through the combination of TypeScript’s built-in codefixes, our own targeted fixes, and deletion of erroneous lines. Compared to existing approaches using linters, our findings suggest that NCC is capable of detecting a larger number of errors per snippet and more error types, and it is more efficient at fixing snippets. We find that 73.7% of the code snippets in NPM documentation have errors; with the use of NCC’s corrections, this number was reduced to 25.1%. Our evaluation confirms that the use of the TypeScript compiler to inform code corrections is a promising strategy to aid in the reuse of code snippets from online sources.

I. INTRODUCTION

Most code snippets online do not run; existing work has shown that only 15.2% of Node.js snippets in NPM package documentation are runnable [1]. Because software developers frequently reuse code from online sources [2], they often need to dedicate time to fixing errors. This introduces challenges when using third-party libraries: examples in documentation are intended to demonstrate usage and non-working snippets can present a barrier to getting started.

Because code snippets are not full, runnable programs with test cases, existing work in automating the detection and fixing of errors has primarily focused on static analysis [3]–[8]. Code reuse tools such as NLP2TestableCode [3] and NCQ [5] combine error detection with heuristic fixes and line deletion to aid developers in reusing snippets. This use of line deletion aims to reduce snippets to an optimal form through a simple deletion operation, looking at errors to determine if the change should be ‘accepted’. Static analysis is also useful for measuring the *quality* of code; existing code reuse tools have made use of parsers, linters and compilers to report errors and find the ‘best’ snippet for a given search query [3], [5]. Additionally, such tools can provide insights on the quality of online code in general: for example, Yang et al. [4] looked at the usability of Stack Overflow snippets via static analysis.

Research in Java leverages the compiler for error detection and correction [3], [9], but JavaScript (and thus the Node.js runtime environment), is an interpreted language that lacks such a compiler. Similar work has instead relied on parsers and linters [4]–[6]. For example, NCQ [5], a command-line REPL

(Read-Eval-Print-Loop) programming environment, which automates the process of reusing code snippets from NPM package documentation, uses ESLint [10] to report errors, and increases the number of snippets without errors from 54.8% to 94.0%. However, these tools serve a different purpose than the compiler (formatting code or generating ASTs), so the errors reported may be more limited; for example, the majority of ESLint rules are stylistic or best practice, not programming errors that affect runnability [6]. Additionally, both ESLint and the SpiderMonkey parser report only a single error if they fail to parse. ESLint needs to successfully parse a snippet to create an AST and run its rule detection. Unlike a compiler, ESLint does not do any type checking. This reveals the need for a better way to evaluate errors in Node.js code.

We investigate how effective the TypeScript [11] compiler is for reporting and fixing errors in Node.js snippets, which is a novel contribution to an area that has otherwise relied on linters and parsers. While TypeScript is a superset of JavaScript with static typing, the compiler is used in VS-Code to provide error highlighting and fix suggestions for JavaScript as well [12], suggesting it may be more useful for error detection and correction than existing approaches. The existing in-editor implementation requires a degree of manual interaction to handle errors; we take TypeScript’s fix suggestions and apply them automatically on given snippets. Furthermore, we implement a limited set of heuristic fixes targeting the most common errors, leveraging TypeScript’s ability to generate ASTs and provide type information. We present our approach, Node Code Correction (NCC), which adapts NCQ’s corrections to use the TypeScript compiler in place of ESLint, including targeted fixes and line deletion. We run both approaches with a dataset of more than two million NPM code snippets and then evaluate NCC against a dataset of Stack Overflow snippet edit pairs representing manual error corrections over time. We report the following findings:

- * The TypeScript compiler reports more errors than ESLint: on average 6.8 vs 1.3 errors per snippet. ESLint reports a single error and no AST for 47.46% of erroneous snippets.
- * TypeScript enables NCC to improve the rate of error-free snippets by 184.67% compared to 72.60% for NCQ, with less empty snippets (7.41% vs 14.33% of the dataset).
- * ESLint’s built-in fixes had a negligible impact on NCQ’s code corrections; only 1 snippet was made error-free. In

contrast, TypeScript’s codefixes corrected 79,613 snippets. * 1,099 (6.88%) of 15,969 Stack Overflow snippets were manually made error free between versions; in comparison, NCC was able to correct 46.77%. Of this 1,099 that were fixed manually, NCC could fix 66.06%.

These results provide evidence that the TypeScript compiler can be useful in automatically identifying and fixing errors, to help reuse online code snippets. We conjecture that further improvement to heuristic fixes can increase the number of corrected snippets. Our approach and related data are available at: <https://doi.org/10.5281/zenodo.8272874>

II. MOTIVATING EXAMPLE

A developer wants to read some data from a URL in Node.js. Let us say that the developer comes across the snippet in Figure 1 (an unedited Stack Overflow snippet from our dataset) while searching on Google. Like many snippets found online, it has errors.

```

1 http.get(url, function(res) {
2   var data = '';
3   res.on('data', function(chunk){data+= chunk;});
4   res.on('end', function(){
5     console.log("BODY: " + data);})
6 }).on('error', function(e) {
7   console.log("Got error: " + e.message);});
8 };

```

Figure 1. Example code snippet from Stack Overflow answer 45582298.

The developer pastes the snippet into their file, but it fails to run with the error ‘SyntaxError: Unexpected token ’}’’, due to a hanging bracket. Furthermore, the TypeScript compiler identifies a number of other issues with the snippet: the identifier `url`, and `http` are also undefined. As we find in Section VI, these are common errors, as example code often omits parts to simplify the snippet.

```

1 + const http = require("http");
2 + var url = "Your Value Here"; // Suggested Type:
3 + string | RequestOptions | URL
4 http.get(url, function(res) {
5   var data = '';
6   res.on('data', function(chunk){data+= chunk;});
7   res.on('end', function(){
8     console.log("BODY: " + data);})
9 }).on('error', function(e) {
10   console.log("Got error: " + e.message);});
11 //};

```

Figure 2. Code snippet after NCC’s corrections.

However, running NCC before using the snippet results in a snippet that reports no errors, as shown in Figure 2. Using TypeScript, NCC detects these errors before without needing to run untrusted code. Using custom fixes, NCC adds the missing `http` require statement; then for the undefined `url`, a placeholder value is declared with suggested types to guide the developer. Line deletion then removes the hanging bracket. From this snippet, the developer can make the necessary changes needed to make the snippet runnable.

We can compare these changes on the original, erroneous version of the snippet, to the manually fixed snippet from our dataset. Similarly, the manually corrected snippet adds the missing `url` variable, which is a string URL. It also adds a

```

1 + var https = require('https');
2 + var url = 'https://www.alphavantage.co/query...';
3 + exports.handler = function (event, context) {
4   \ https.get(url, function(res) {
5     ...});
6 };

```

Figure 3. Excerpt of the manually corrected snippet.

`require`, but changes the library to `https` to match the URL. To correct the hanging bracket, the code has been wrapped in an exported function.

In contrast, ESLint reports only a single parsing error for the original snippet, and so the only change is to comment out the bracket. This motivating example illustrates how the capabilities of the TypeScript compiler can be used to help with corrections that benefit developer workflows.

III. RELATED WORK

While many automated code reuse tools allow developers to find and use snippets online from within their programming environment, developers still need to spend time correcting them when they do not work. Node Code Correction combines ideas from three areas of work into one tool, in order to help developers reuse code from online: 1) error detection via static analysis; 2) code correction and 3) code deletion. We discuss existing work in these areas, both in the limited JavaScript and Node.js space, and in other languages.

A. Static Analysis

Previous work has looked at error detection in JavaScript and Node.js, using parsers [4], linters [5], [6] or runtime errors [1], [4]. The benefit of static analysis is that it can report multiple errors, is typically fast, and that code can be evaluated without running it; this is especially useful when most online snippets do not run [1], [4]. Additionally, it is undesirable for a code reuse tool to run arbitrary code from online, when snippets can be malicious or contain vulnerabilities. For these reasons, static analysis can be useful for providing information about large sets of code, or for on-demand use in an automated code reuse pipeline. For example, tools such as NCQ [5] and NLP2TestableCode [3] use errors to inform fixes and recommend the highest quality snippets first. However, most of the issues that linters like ESLint report are stylistic; Campos et al. ran the standard ESLint configuration on JavaScript code snippets mined from Stack Overflow and found that no snippets were free of rule violations, but that 163 rules could be characterised as ‘stylistic issues’ or ‘best practice’ [6].

Similar work in Java has made use of static analysis tools like PMD [13] and compilers to detect and correct errors in code snippets [3], [7]–[9]. The process of converting code into another lower-level language is more complicated than just generating an AST (compilers parse code as only one step of the compilation process), meaning that they report errors that parsing alone does not. Because Java code must be compiled before it can be run, the ability to compile a snippet is a useful measure of quality in a reuse context – code that does not compile is thus not runnable. Many compilers, such as

the Eclipse Java compiler and the TypeScript compiler, are designed to report multiple errors and are used to report error information within an IDE. To the best of our knowledge, no existing work attempts to use the TypeScript compiler to evaluate and fix errors in Node.js code. Based on these observations, we devise NCC to fill this gap.

B. Automatic Code Correction

Much work on fixing code has focused on software bugs in runnable programs, evaluated via test cases. In contrast to this, code correction in the context of code reuse deals with fragmented, often unrunnable code, where these approaches cannot be applied. To solve the problem of correcting unrunnable snippets, existing tools rely on static analysis to identify errors and inform heuristic fixes. CSnippEx [9], for Java, employs an existing suite of fixes from Eclipse, while NCQ [5] does the same in Node.js using ESLint’s fixes. NLP2TestableCode [3] in Java uses a set of custom heuristic fixes, and is able to insert missing import statements and variable definitions. Jigsaw [14], another Java tool, allows developers to supply a method to integrate and a destination class or function, then extracts structural information to make integration changes. Where it cannot automatically fix integration errors, it inserts comments and highlights parts of code for developer attention.

Besides NCQ, other work in JavaScript and Node.js looks at repairing software bugs in runnable code. Vejovis [15] automatically suggests repairs for DOM-based JavaScript faults to developers, but these repairs require runnable code and are not applicable for Node.js. The use of AI models to fix code is also of interest: Lajkó et al. [16] look at the use of the GPT-2 model to fix software bugs; after training the model to fix JavaScript bugs, they found that it did so correctly in most cases. AI tools that generate code snippets, such as GitHub Copilot [17], a plug-in for VSCode that uses OpenAI’s more advanced GPT-3-based Codex [18], are able to generate snippets that match the surrounding context, eliminating the need to integrate snippets. However, there is some concern about the quality of the output of these systems, with regard to bugs, vulnerabilities, and correctness for given queries [19], [20]. NCC aims to build on existing work on correcting errors in Node.js code snippets, by combining TypeScript’s existing fix suite, with custom heuristic fixes, as well as utilising comments where developer intervention is still needed. Additionally, we hypothesise that better error detection using a compiler will enable more accurate error correction.

C. Code Deletion

Code deletion, for example, at the granularity of lines or statements, is a unary operator that is easy to implement. It also does not require any code analysis or synthesis and can be a component of a more complex operation, such as replacing a line with another. Therefore, code deletion is typically included in studies related to code improvement [7], [8], [21]–[24]. Often, these studies report errors that are *fixed* (relative to a given test suite) by removing the offending code.

For the problem of correcting unrunnable code where test cases cannot be used, line deletion, in combination with error reporting, has previously been used as a solution. For example, NLP2TestableCode [3], which is an Eclipse plug-in that assists in reuse of Stack Overflow code, employs a line deletion algorithm as the last step in a suite of fixes, using the Eclipse compiler to provide error information. The NCQ code reuse tool [5] implements a similar functionality for Node.js, using ESLint to evaluate errors. Similarly, Licorish and Wagner [7] combined static analysis with the Gin genetic improvement framework [25] (which includes deletion operations, among others) to improve Java code snippets on Stack Overflow. In contrast to these works, we investigated the possibility of using the TypeScript compiler to inform line deletions as just one of the potential ways it could be used for code corrections.

IV. APPROACH

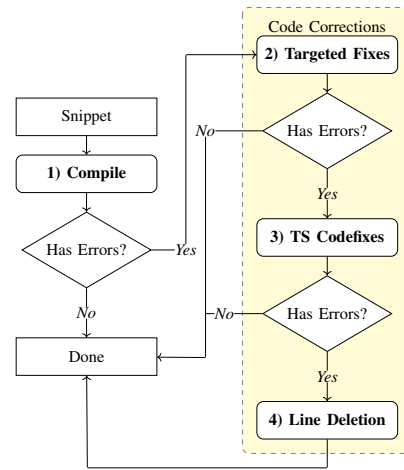


Figure 4. The NCC pipeline.

Node Code Correction (NCC) has four stages, illustrated in Figure 4; 1) compilation to identify errors; 2) targeted fixes; 3) TypeScript codefixes and finally 4) line deletion. Each snippet is initially compiled to check for errors; then, for erroneous snippets, the code correction process begins. First, a series of heuristic custom fixes are attempted; if errors continue to exist, then TypeScript’s builtin codefixes are applied where available. Finally, line deletion is employed to handle remaining errors. After any change, the code is compiled again to update the error information. This section describes each aspect of NCC.

A. Identifying Errors

Identifying errors is the first step to correcting errors. To do this, NCC uses the TypeScript compiler from version 4.9.4 of the TypeScript package. To optimise compilation speed, the TypeScript compiler is run programmatically and in-memory for a given string of code, using a custom `CompilerHost` that handles the interface between the compiler and the ‘file system’. This custom `CompilerHost` stores the code string as an in-memory `sourceFile`, instead of looking on the file system. Furthermore, we cache any required files (such as

TypeScript definition files) between compiles. By default, the TypeScript compiler will check all loaded files for errors, considerably slowing down the compilation time, so we specify only looking at our single input file. The TypeScript compiler is then isolated from other code, by only allowing it to access files in the ‘typescript’ and ‘@types/node’ folders that are needed for TypeScript to function.

We configured the compiler with options to match a default v18.16.0 Node.js environment for example, we enabled Node.js types and did not allow JSX (JavaScript XML) because the dataset is meant to be Node.js code only. To trigger the TypeScript compiler’s JavaScript mode, the in-memory file was named with a ‘.js’ file ending. Due to the size of the dataset, for this investigation of the entire NPM registry, we do not try to install each snippet’s source package, though the TypeScript compiler is capable of deriving additional type information that could have enabled more accurate type information. Additionally, the need to install packages for each snippet would increase the time to fix, which is one of the benefits of static analysis like this. We simply ignore the ‘Cannot find module’ error on `require` statements, and the compiler will then continue to generate general Node.js errors. On compilation, the compiler generates a list of diagnostics, including error code, message, start location, and length. To deal with rare cases where the compiler threw an error or never finished compiling, we run the compiler in a separate process with a timeout of 60 seconds.

B. Targeted Fixes

On the basis of our experimentation with prototype versions of NCC, we devise a series of custom heuristic fixes that address common errors that other stages cannot correct. Our heuristics for identifying and correcting errors thus embody a series of iterative enhancements that integrate lessons learned from these early prototypes.

We created two fixes for the common error `Cannot find name`, which occurs for undeclared variables. We identify that the cause of the error is either a missing ‘require’ for a package, or a variable not being defined. Our two fixes thus are to insert the import or define a placeholder variable. We leverage the TypeScript compiler’s ability to generate ASTs for even erroneous code to provide information about the context and surrounding code, and to keep track of errors that exist on the same line as each other. We ignore cases of the `Cannot find name` error where it may be reported for non-code (for example, terminal commands): in cases of `expression expected` and `unexpected keyword or identifier` we presume that the code on that line has additional issues, and we make no changes.

For missing `require` statements, we check if the identifier could be an API usage, and then check if the name matches to a built-in library. For these cases, we insert a `require` statement. Undefined functions are ignored to allow TypeScript’s codefixes to handle these cases instead. In other, non-function, cases, we attempt to get the expected type of the undeclared variable. That is: if the identifier is an

argument of a function, we check for the expected type from the parent function. From here, we can insert a placeholder string, number or array of strings or numbers. Where a type cannot be determined, we default to a string. Additionally, for more complex types, we default to a placeholder string with a comment noting the suggested type. These placeholders serve to move the snippet to a more ‘correct’ state, while indicating where developer intervention may be needed. The motivating example in Section II demonstrates a case in which both of these fixes are applied. After applying a fix, we compile to see if the changes do not increase the total number of errors; if not, the change is kept.

C. TypeScript Codefixes

We employ TypeScript’s codefixes (the Quick Fix suggestions that TypeScript can provide to an IDE, for example, when integrated with VSCode) to automatically correct errors. TypeScript codefixes require use of the `LanguageService` API, not the compiler, but, similarly to the TypeScript compiler, we speed up runs via in-memory objects and caching. Sharing a `DocumentRegistry` object between runs, and only updating the input ‘file’ for each snippet, gives considerable speed benefits. TypeScript supports fixes for 1,190 of its 1,878 available error types.

We adapt the codefix procedure of the Microsoft `ts-fix` tool [26], which automatically fixes errors in TypeScript projects. For each error, a set of `CodeFixActions` is supplied if they exist, each with its own set of changes that must be made to the text. All possible changes are combined into a list, sorted by the earliest start and then the smallest change. Then, we filter the list to remove changes that would overlap (i.e., affect the same part of the string), before applying them to the text. Then we compile the code again to update the error count.

D. Line Deletion

Line deletion is a commonly used technique to reduce errors in code snippets [5], [7], [8], [21]–[24]. We adapt the NCQ line deletion algorithm to work with the TypeScript compiler. The deletion algorithm functions as illustrated in Algorithm 1, and is run on snippets that still have errors after the codefix stage. The algorithm attempts to find the ‘best’ snippet based on error count, by deleting lines affected by errors. The ‘deletion’ occurs by commenting out the line, just like in NCQ; in a code reuse situation these erroneous lines may still be useful to a developer by providing additional context, and commented out code can aid developers in fixing bugs, debugging and adding features [27]. We prefer line deletion over statement deletion for the issue of code fragments, as not all snippets are parsable.

First, the snippet S is compiled to find errors. If there are no errors, the process stops there. If there are errors, the algorithm starts with the first error and attempts to delete the associated line. The new snippet is then re-evaluated, and if the error count did not increase, the deletion is kept and the `errorNo` variable is reset as the error list is now changed. If the change made things worse, we revert the change and move on to the

Algorithm 1: Line Deletion Algorithm

```
Sbest ← Initial snippet
// Step 1: Get errors
Sbest.errors ← Compile(Sbest)
done ← false
errorNo ← 0
while done == false do
  Scurrent ← Sbest
  // Step 2a: Check if done
  if errorNo ≥ Scurrent.errors.length then
    | done ← true
  // Step 2b: Try delete error
  else
    Scurrent.DeleteLineFor(errorNo)
    Scurrent.errors ← Compile(Scurrent)
    // Step 4a: Keep deletion
    if Scurrent.errors ≤ Sbest.Errors then
      | Sbest ← Scurrent
      | errorNo ← 0
    // Step 4b: Try next error
    else
      | errorNo ++
return Sbest
```

next error. The loop ends when there are no errors or all errors have been processed, and the algorithm returns the snippet with the least errors it can produce. In some cases, snippets are commented out completely (so-called ‘empty’ snippets) in order to reduce the snippet to zero errors.

The major change from NCQ is that TypeScript reports more than a single ‘failed here’ parsing error, unlike ESLint. This means that NCC’s line deletion algorithm is capable of trying multiple changes when one does not work. Furthermore, because the mined dataset still contains some non-Node.js snippets even after filtering, we handle additional edge cases not shown in the algorithm based on the unexpected behaviour of the TypeScript compiler. We ignore the previously discussed crashing snippets. Additionally, it is possible for an error to persist even on a commented-out line, so we check if the line has been commented out and skip it. In cases where the reported error location exceeds the actual snippet length, which we interpret as a problem parsing the snippet, we terminate the line deletion process.

V. DATASET

This section provides an overview of the dataset used to evaluate the performance of the code correction tool. The dataset consists of two main sources: NPM snippets and Stack Overflow edits, each described in detail in their respective subsections. ‘Snippet’ in both instances refers to code fragments mined directly from either markdown or HTML, by looking for code blocks. We do not attempt any combination of related snippets in a single source, as developers and tools often treat snippets as self-contained, even when that may not be true.

A. NPM Snippets

Reid et al. [5] originally ran NCQ’s code corrections over a dataset of 2,161,911 code snippets mined from the NPM registry as of May 2021. The dataset contains snippets extracted

from markdown code blocks in the package READMEs. Heuristics were employed to ensure the dataset was filtered for only Node.js snippets, manually verified on a sample of 384 READMEs (confidence level 95%, confidence interval 5). However, non-JavaScript code snippets (including terminal commands, TypeScript and JSX, a JavaScript extension used for React) may still be present within the dataset.

For our evaluation, we use the same publicly available dataset for our evaluation. However, because NCQ is a Node.js REPL, it implements some REPL-specific rules and fixes to make reusing code snippets in this environment easier. Because of this, we rerun its correction on the dataset after disabling these rules to better emulate the scenario described in Section II (a developer looking to reuse code snippets in a regular Node.js programming environment) and report errors before and after fixes.

B. Stack Overflow Edits

In order to compare NCC’s performance to how developers manually edit code, we evaluate on a set of Stack Overflow snippets, for which we have the first and most recent edit. We used the December 2020 version of the SOTorrent dataset [28], [29], retrieving the code-only `PostBlockVersions` for all accepted answers of posts tagged ‘Node.js’, giving us a total of 299,389 snippet versions, or ‘edits’, across 182,205 snippets. For our SOEdits dataset, we look only at snippets where there are at least two versions, there was some change between the first and last version, and the first version has at least one error, creating a dataset of 21,431 snippet ‘before’ and ‘after’ edit pairs. These pairs represent an original erroneous snippet, and the current, edited snippet on SO.

Table I
SUMMARY OF SOEDITS DATASET.

All snippets	182,205
All versions	299,389
All SOEdit pairs	21,431
Improvement only	15,969
Fixed only	1,099

By running the TypeScript compiler, we observe that this set of snippet pairs does not necessarily represent an improvement over time; overall, the number of errors increased between edit pairs, as did the number of lines of code. For 74.51% of SO edit pairs (15,969 pairs), there was an improvement in errors between edits, and for only 5.12% of edits (1,099 pairs), all errors were corrected. For this reason, we further filter the dataset to the 15,969 snippets that show improvement and create an additional subset for the 1,099 snippets that were ‘fixed’. Table I shows the breakdown of the data.

VI. EVALUATION

We run both NCQ and NCC’s corrections on a dataset of 2,161,911 code snippets from NPM package documentation (described in Section V-A) and record results at each stage. To

establish baseline data of what errors ESLint and TypeScript can identify, we also run only the error reporting. Experiments were run with the latest LTS version of Node.js as of April 2023 (18.16.0), version 4.9.4 of TypeScript and version 8.31.0 of ESLint. We make the assumption that developers looking for code online expect it to be up-to-date and compatible with the recommended version of Node.js. We configure the error reporters in NCQ and NCC (ESLint and TypeScript) comparatively to emulate the scenario described in Section II; a code snippet pasted into an empty file, in an otherwise empty Node.js project, with no packages installed. Additionally, our error reporters are configured for CommonJS, or ‘script’ mode, where `require` statements are used to import packages and top-level `await` is not allowed. Because NCQ’s corrections were designed for its REPL context, REPL-specific rules and fixes were disabled so as not to impact results. Although the dataset may still contain non-Node.js code snippets despite filtering, such as TypeScript and JSX, we limit the evaluation to Node.js; where TS and ESLint have options to process this code without errors, we do not enable them. We ask the following research questions:

- RQ1.** What errors does TypeScript detect in NPM documentation?
- RQ2.** How does error detection differ between ESLint and TypeScript?
- RQ3.** What is the impact of NCC on the set of NPM snippets?
- RQ4.** How does NCC compare to NCQ’s code corrections?

Furthermore, to evaluate the NCC results against the way developers manually fix errors, we compare the results with the set of improvements in the SOEdits dataset, described in Section V-B. We ask the following research question:

- RQ5.** How does NCC compare to manual fixes?

A. What errors does TypeScript detect in NPM documentation?

We ask this question to characterise the frequency and types of errors in NPM package documentation and also to establish a baseline to compare our corrections. We ran the TypeScript compiler on all 2,161,911 code snippets and found that only 569,201 code snippets (26.3%) had no errors. TypeScript identified a total of 14,707,149 errors in the set, an average of 6.8 errors per snippet. Looking at only erroneous snippets, the average number increases to 9.2.

TypeScript reports 404 different error types on our dataset. Almost half of the 14.7 million errors TypeScript detects are for the error type `cannot find name`, with 7.2 million occurrences; this is visible in Figure 5. This error reports cases where an identifier was referenced without a declaration. There is a similar, but separately numbered, error that suggests an alternative name, where a misspelling is suspected, accounting for another 198,413 errors. The second most common error type, `character expected`, accounts for 1.6 million errors. With the exception of the `JSX` error which appeared

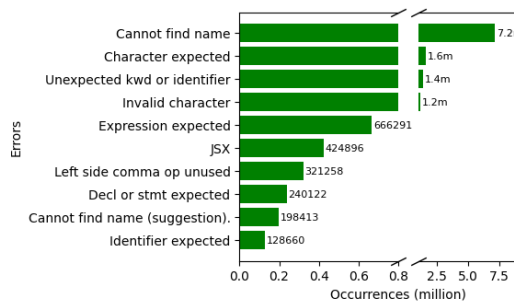


Figure 5. Most common error types in NPM documentation, reported by TS.

in 154,737 code snippets (7.16%), common errors can be characterised as missing or unexpected characters, keywords, identifiers, statements, or expressions. Error messages typically provide the error-causing token or the expected token. TypeScript is also able to detect when functions do not exist on a type; the `Property` does not exist on type error is the 12th most common with 83,483 occurrences.

```

1 var prompt = require('prompt');
2
3 prompt.start();
4 prompt.get(['username', 'email'], function (err, result
5   ) {
6     console.log('Command-line input received:')
7     console.log('  username: ' + result.username)
8     console.log('  email: ' + result.email);
9   });
10
11 const {username, email} = await prompt.get(['username',
12   'email']);

```

Figure 6. Two code snippets from the package `prompt`.

Figure 6 shows a common situation in NPM package documentation. These two code snippets from the `prompt` package demonstrate two ways to get input from a user on the command-line using a prompt; using a callback or using `await`. However, the second code snippet would generate a `cannot find name` error when evaluated by the TypeScript compiler. The variable `prompt` is undeclared in the second snippet, but not in the first.

Code snippets are often not intended to be working examples but rather to demonstrate functionality; they often omit code that would be repeated between code snippets, such as the `require` statement in Figure 6. However, developers and automated tools still use them this way. The number of `cannot find name` errors in the dataset suggests that missing variables are common and that this practice is widespread.

Summary: The majority (73.7%) of code snippets in NPM package documentation have some kind of error. On average, the snippets have 6.8 errors. The most common error was for undeclared variables.

B. How does error detection differ between ESLint and TypeScript?

To compare the two error reporters, we ran ESLint on the same snippets. We use a modified version of the configuration

from NCQ, with the REPL specific errors disabled. We disable ‘linting’ rules concerning formatting and only look at errors that would affect code functionality.

ESLint reports a similar rate of erroneous snippets, with only 26.3% of snippets having no errors. However, we observe that the average number of errors per snippet sits at 1.26 errors (and 1.71 errors for the erroneous set). In fact, ESLint reports only a fraction (18.5%) of the errors that TypeScript can on the same set. This is because of the 2,722,241 errors reported, 27.8% are parsing errors.

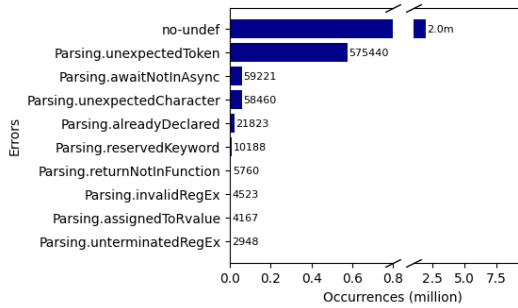


Figure 7. The 10 most common error types via ESLint.

ESLint reports 185 different error types, of which 175 of the types (94.6%) are parsing errors. Figure 7 reflects this, where all but the most common error reported by ESLint (no-undef at 2m occurrences) are parsing errors, such as unexpected tokens (575,440 occurrences) and use of await outside of an async function (only allowed in ES modules), at 59,221 instances. The prevalence of parsing errors is an issue for two reasons. First, ESLint reports only a single parsing error per snippet indicating why parsing failed, thus these snippets do not generate an AST, nor can ESLint run its rule detection or fixes. This means for 47.46% of snippets with errors, we are only able to detect a single, unfixable error. Secondly, ESLint rule detection besides from ‘no-undef’ accounts for only 2,211 errors. TypeScript’s ‘unexpected token’ error, for example, occurs for only 60,885 occurrences; instead, TypeScript has an increase in other more specific error types that might provide more useful information on the cause of the error. The types of errors ESLint can report are limited due to its intended purpose as a linter; further lowering the error rate, the majority of rules are not enabled by NCQ as they are not useful for identifying erroneous code. In this context Thus, the error information is incomplete, and ESLint may not necessarily be useful for evaluating runnability or informing fixes.

Summary: ESLint reports considerably fewer errors than the TypeScript compiler – an average of 1.3 vs. 6.8 per snippet. 47.46% of erroneous snippets have a single error where parsing failed, resulting in an average of 1.71 errors per erroneous snippet compared to TypeScript’s 9.2. In these cases, it also cannot generate an AST to enable fixes. These results indicate that ESLint is limited in what it can tell us about code.

C. What is the impact of NCC on the set of NPM snippets?

First, we look at the impact of only the TypeScript codefixes on the set of snippets. After applying the codefixes, the number of snippets without errors increased from 569,201 (26.3%) to 648,814 (30.0%). The total number of errors was reduced from 14,707,149 to 14,096,112 (a decrease of 4.2%). In total, 602,629 snippets (27.9%) had changes made to fix errors.

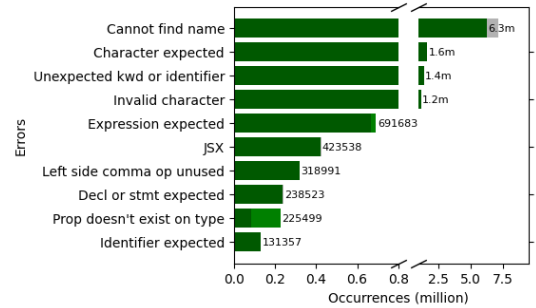


Figure 8. The 10 most common error types after TS codefixes. Light Grey represents a decrease in errors from previous results in Figure 5, where light green represents an increase.

Figure 8 illustrates the most common error types after codefixes and shows significant changes. All 10 of the most common errors in Figure 5 had fixes, however, not all fixes can be applied to every error, and some fixes can introduce new errors. The most common error is still Cannot find name, but it has reduced from 7.2 million to 6.3 million occurrences. The error type Cannot find name (suggestion) no longer appears in the most common errors, reducing from 198,313 occurrences to just 25,979. As discussed in Section VI-A, this error type directly relates to its suggested fix, so the reduction is logical here. However, the error type Property doesn’t exist on type increased by 142,016 occurrences. Expression expected also visibly increased. The other errors have minor increases/decreases that are not visible on this scale. The increase in error Property doesn’t exist on type probably results from fixes for undefined variables, where a variable is defined that does not make sense on the basis of usage.

Next, we consider the impact of line deletion in combination with TS codefixes. We find that the number of snippets without errors increased from the original 569,201 (26.3%) to 1,622,272 (75.0%), an increase of 185.0%. The total number of errors that could not be fixed also decreased from the original 14,707,149 to 925,277. The average number of errors per snippet decreased to 0.43. In total, 1,343,992 snippets (62.2% of the total) had changes made to reduce errors.

The line deletion stage accounts for a 150.0% improvement, or additional 973,458 error-free snippets, from the TypeScript codefix stage. However, it also comments out all lines for 483,169 code snippets (22.3% of the total set and 49.6% of snippets it makes error-free). To measure the impact of the line deletion algorithm, we counted the number of lines before and after deletion. In total, 4,031,366 lines of code were com-

mented out, 22.1% of the total lines of code. Combined with the lines added from TypeScript codefixes, there are 2,902,083 fewer lines after NCC’s corrections. However, deleted lines are only commented out, so they can still be useful to developers by providing additional context and guiding them with what to do next.

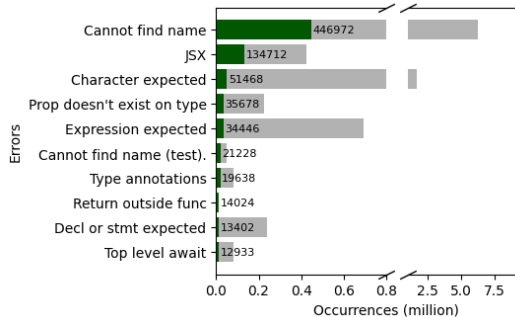


Figure 9. The 10 most common error types after deletion and TS codefixes. Shading represents improvement over Figure 8.

The types of errors that were reported changed considerably between codefixes and deletion, as seen in Figure 9. The original 7.2 million instances of `cannot find name` were again reduced to 446,972 instances. New errors now populate the top 10: `cannot find name (it)` where a testing library using the function `it()` was not installed, `Type annotations` for the use of TypeScript, and `Top level await`. All common errors saw reductions from their previous values, except `Return outside function`, which increased by 7,728 occurrences after line deletion.

```

1 + var s = "YOUR VALUE HERE";
2 var words = s.split(" ");

```

Figure 10. Example of undeclared variable with no fix, and a proposed fix.

Based on these results, we implement a limited suite of targeted fixes for the most common error that still persists: `cannot find name`. We conjecture that the ability to define variables will enable NCC to reduce the number of line deletions, and thus empty snippets. Despite the error type `Cannot find name` having TS codefixes in some cases, seemingly ‘simple’ cases such as the example in Figure 10 cannot be fixed. On the basis of these cases, we introduce custom fixes.

We run all fixes on the dataset and observe that the number of snippets with no errors decreases slightly by 1,929 (a decrease from 75.0% to 74.94%). However, this does not tell the entire story: the total number of errors fell by 2.76% to 899,774, and the number of empty snippets fell considerably from 22.35% of the dataset to 7.41%. Figure 11 shows how some errors increase but that there is a considerable decrease in `Cannot find name`. We see an increase in the error `property doesn't exist on type`, likely due to the addition of placeholder definitions which default to strings in many cases. Similarly, the increasing error `expression`

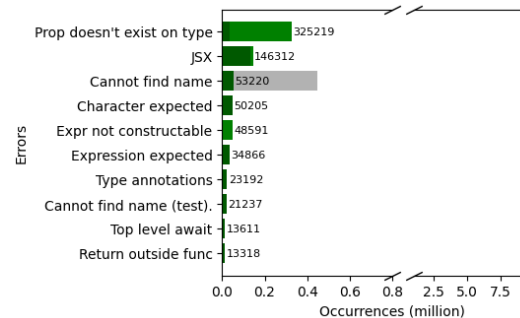


Figure 11. The 10 most common error types after all fixes. Shading represents prior results from Figure 9.

`not constructable` also handles a similar case, where the previously undefined identifier should instead be a constructable object.

Summary: NCC improves the number of error-free snippets by 184.67%, and most of the remaining erroneous snippets have some changes to reduce errors. However, 7.41% of the snippets are entirely commented out by the corrections. The results indicate that leveraging TypeScript enables NCC’s custom fixes to decrease errors, but that additional heuristic fixes could further reduce the reliance on ‘last resort’ line deletion.

D. How does NCC compare to NCQ’s code corrections?

We ask this question to investigate whether the use of a compiler like TypeScript instead of a linter can improve error-informed code corrections. NCQ’s code correction approach consists of three components: evaluating errors using ESLint, fixing errors using ESLint’s built-in fixes, and the line deletion algorithm which runs on parsing errors. We record errors before and after all correction steps.

With all fixes, we find that NCQ’s corrections are able to increase the number of snippets without errors from 569,419 to 982,832 (45.46% of total code snippets), an increase of 413,413 code snippets (72.6%). Furthermore, the total number of reported errors (that could not be fixed) actually increases by 34,502, as the correction of parsing errors enable more ESLint rule violations to be detected. In line with this, the average errors per erroneous snippet increased to 2.34. 113,621 of the 413,413 snippets made error-free (27.48%) had all lines commented out, and the line deletion algorithm removed 3,461,047 lines (20.2% of all lines).

Figure 12 shows the most common errors after fixes. Errors reduced for parsing errors as expected, with `unexpected tokens` now at 123,695 occurrences. Now that so many parsing errors have been corrected with the deletion algorithm, other errors appear in the top 10. We see that `no-undef` saw an increase to 2,626,174 occurrences, as NCQ employs no fixes for this error, and other non-parsing errors are now visible such as `no-const-assign`. We note from our results that

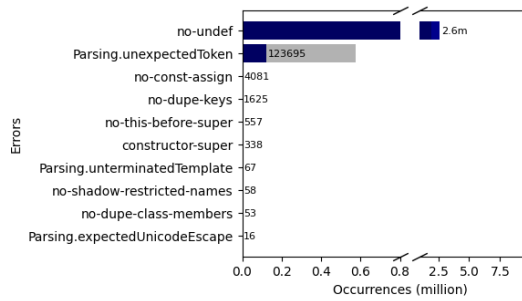


Figure 12. The 10 most common error types via ESLint after NCQ’s corrections. Shading represents the initial values before fixes (see Figure 7).

ESLint’s built-in fixes have very little impact on the dataset: after the ESLint fix stage, errors only reduced by 2 and a single snippet was made error-free. ESLint’s fixes mostly solve formatting issues and only work on parsable snippets, so this is expected. We compare these results with NCC’s use of TypeScript codefixes, which corrected all errors in 79,613 snippets and resulted in a 611,037 error reduction.

It is difficult to compare the impact of each approach on the quality of code snippets; we do not attempt to run snippets after fixes to see if there is an improvement in runnability, and neither evaluator can give us a ‘correct’ number of errors. We also cannot compare the exact number of errors between approaches, as each evaluator reports errors differently. However, we can see that NCQ results in more empty snippets as part of its corrections, and that it does not correct as many snippets. Because most erroneous snippets with ESLint have a single parsing error, this means that if a line deletion does not improve the code snippet, there are no alternatives to try. Empty snippets and number of deleted lines also reduced between approaches: 14.29% of corrected snippets were empty for NCC (7.41% of the dataset), compared to NCQ’s 27.48% (14.33%). This suggests that the use of TypeScript enables more accurate line deletion and that additional fixes reduce the reliance on deletion.

Summary: Compared to NCQ’s code corrections, we find that NCC has a higher improvement rate and that NCC leaves fewer snippets empty (7.41% vs 14.33%). We find that ESLint’s automated fixes implemented in NCQ had little effect on improving code snippets, only fixing 1 snippet.

E. How does NCC compare to manual fixes?

To compare NCC to manual fixes, we evaluate against Stack Overflow snippet pairs, for which we have an original snippet containing at least one error (the ‘pre-edit’ snippet), and the most recent with reduced errors (the ‘post-edit’ snippet). We look at both the total ‘improvement’ set, and the 1,099 subset where all errors were fixed. Though we do not expect our limited suite of fixes to correct all 1,099 snippets, the aim of the comparison is to see how well NCC performs despite this.

First, we observe the error landscape of the pre-edit ‘improvement’ set. The TypeScript compiler reports a total of 160,602 errors in the dataset: an average of 10.06 errors per snippet. This set of snippets only contains erroneous snippets, so all snippets have at least one error. Figure 13 shows the most common error types before editing, with the most common error remaining ‘cannot find name’ like in the previous results. Again, Stack Overflow snippets often miss elements that might exist in other snippets or in the question.

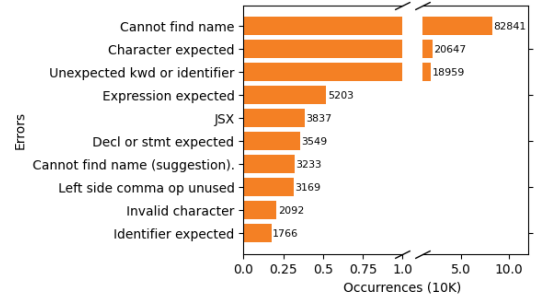


Figure 13. The 10 most common errors for Stack Overflow Edits pre-edit.

Next, we observe the reduction in errors after manual fixes, represented in the ‘post-edit’ set. We see a 17.60% reduction in total errors and 6.88% of the snippets have all errors corrected. Figure 14 shows the change in error types, which can be summarised as a general reduction in all common error types.

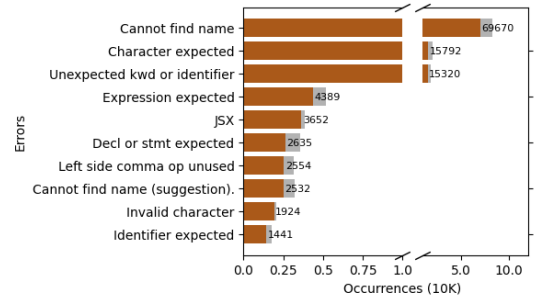


Figure 14. The 10 most common error types for Stack Overflow Edits post-edit. Grey represents the change between edits.

When we run NCC over the original snippets, we observe that our suite of fixes enables a 90.69% decrease in errors. 7,469 snippets are made error-free: an additional 6,370 snippets over the post-manual-edit set, representing 46.77% of the dataset. This result is achieved with a 5.71% rate of empty (‘commented out’) snippets, compared to only 3 (0.27%) for manual edits. Figure 15 shows how the error landscape changes: The occurrences of cannot find name, previously the most common error, reduce to only 1,281 occurrences. Again, we see similar errors increase as in Section VI-C. However, we check for an increase in errors after making changes to ensure that the change does not make the code worse, and the value is intended to be modified by developers with their own value.

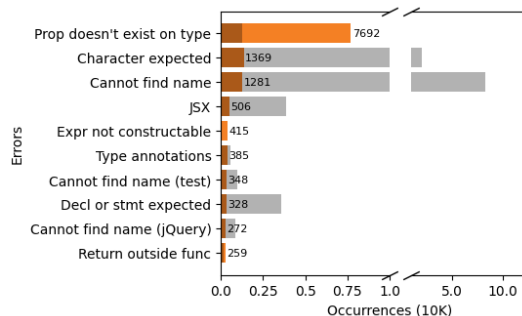


Figure 15. The most common errors for Stack Overflow snippets after NCC.

For the subset of 1,099 snippets that were manually fixed in their post-edit version, NCC is able to fix 726, 66.06% of the set. 47 (4.28%) of the snippets are entirely commented out by NCC’s fixes.

Summary: NCC can resolve all errors for 46.77% of SO snippets, reducing errors by 90.69% with a rate of 5.71% empty snippets. Evaluated against snippets with manual corrections, NCC can fix 66.06% of these snippets, which is a promising result.

VII. THREATS AND LIMITATIONS

There are several potential threats to the validity of this study. Firstly, our results on quality and correctness of the code snippets are based on reported errors from both TypeScript and ESLint, but neither of these tools can accurately represent the *runnability* of the code. Furthermore, we did not try to install each package within our dataset, but because TypeScript can gather type information and use it to report errors, this could have provided additional error information. We assume that the inability to parse or compile a snippet relates to its quality and runnability, which is true for compilable languages like Java but may not necessarily hold for Node.js. Similarly, because we do not run code snippets, either before or after fixes, we cannot know the impact of fixes on runnability. Our fixes for Stack Overflow snippets may report fewer errors, but our automatic fixes may not be similar to the kinds of fixes that developers produce manually. Additionally, removing lines may reduce errors at the cost of expected behaviour. Because we focus on lines, and not statements, errors over multiple lines may not easily be addressed by our algorithm.

Snippets were mined devoid of context, in order to replicate developer copy-paste and code recommender systems like NCQ, but this method may account for some of the missing variable errors. Additionally, while care was taken to limit the mined datasets to only Node.js code, as described in Section V, non-code still exists in the dataset and may impact results. Finally, the results of our evaluation are specific to Node.js and the version used, and we cannot claim that they generalise to other languages, or even versions of Node.js. Additionally, there are limitations to our approach. NCC

simply re-implements `ts-fix`’s batch approach to applying TypeScript’s fixes, which does not validate each fix individually. Like the line deletion algorithm and heuristic fixes, each change could be checked via the compiler to ensure that no fix makes a snippet worse. Furthermore, heuristic fixes can always be further refined to handle more situations. We acknowledge the limitations of such fixes, in that they must be individually designed for each error case and make guesses about missing parts of code. We look with interest at Large Language Models (LLMs) like OpenAI’s Codex and ChatGPT that might provide new AI solutions for this problem. GitHub’s Copilot plug-in already generates snippets in the editor for a given task and code context and could change code reuse practices. However, there are concerns about the quality of generated code. Future work may investigate how a similar system can be applied to existing code snippets.

VIII. CONCLUSION AND FUTURE WORK

Developers often rely on code snippets found online for reference and assistance in their projects. However, most of these snippets are not runnable, requiring developers to spend additional time fixing errors, which can be especially challenging when using third-party libraries. Existing approaches to automatically identify and fix errors in snippets have primarily focused on static analysis using parsers and linters, as snippets often lack test cases or do not run. Although these techniques have proven useful in some cases, there is still a need for a better way to evaluate and correct errors in Node.js code. Our work aims to address this gap by using the TypeScript compiler for more effective and accurate code correction compared to a linter like ESLint, which on average reports only one error per snippet.

Our results indicate that the TypeScript compiler enables more effective and accurate code identification and correction when compared to ESLint. The TypeScript compiler is also capable of detecting more errors and more informative errors, and its built-in fixes affect more snippets. Additionally, the reported error information and ASTs generated by the TypeScript compiler enable the use of additional heuristic fixes on more snippets. Based on these results, we suggest the use of the TypeScript compiler for static analysis on Node.js datasets over linters, and the NCC approach for automating code reuse.

Future work could integrate NCC within a code recommendation system or as a IDE plug-in, to find, insert and then correct code snippets from online, similar to NCQ. Additionally, we could study how useful developers find the fixes generated by NCC, either by asking them questions about the changes or having them use the tool in a code reuse scenario. It may also be interesting to investigate how accurately TypeScript errors correlate with runnability, when trying to run code snippets, and how well NCC makes the snippets runnable.

ACKNOWLEDGEMENT

Brittany’s research was supported by an Australian Government Research Training Program (RTP) Scholarship.

REFERENCES

- [1] B. Chinthanet, B. Reid, C. Treude, M. Wagner, R. G. Kula, T. Ishio, and K. Matsumoto, "What makes a good node.js package? investigating users, contributors, and runnability," *arXiv preprint arXiv:2106.12239*, 2021. [Online]. Available: <https://doi.org/10.48550/arXiv.2106.12239>
- [2] S. Baltes and S. Diehl, "Usage and attribution of stack overflow code snippets in github projects," *Empirical Software Engineering (EMSE)*, vol. 24, no. 3, pp. 1259–1295, 2019. [Online]. Available: <https://doi.org/10.1007/s10664-018-9650-5>
- [3] B. Reid, C. Treude, and M. Wagner, "Optimising the fit of stack overflow code snippets into existing code," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. GECCO '20. New York, NY, USA: ACM, 2020, p. 1945–1953. [Online]. Available: <https://doi.org/10.1145/3377929.3398087>
- [4] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: An analysis of stack overflow code snippets," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, ser. MSR '16. New York, NY, USA: ACM, 2016, p. 391–402. [Online]. Available: <https://doi.org/10.1145/2901739.2901767>
- [5] B. Reid, M. d'Amorim, M. Wagner, and C. Treude, "Ncq: Code reuse support for node.js developers," *IEEE Transactions on Software Engineering (TSE)*, vol. 49, no. 5, pp. 3205–3225, 2023.
- [6] U. Campos, G. Smethurst, J. P. Moraes, R. Bonifácio, and G. Pinto, "Mining rule violations in javascript code snippets," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, M. D. Storey, B. Adams, and S. Haiduc, Eds. IEEE / ACM, 2019, pp. 195–199. [Online]. Available: <https://doi.org/10.1109/MSR.2019.00039>
- [7] S. A. Licorish and M. Wagner, "Combining gin and pmd for code improvements," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. GECCO '22. New York, NY, USA: ACM, 2022, p. 790–793. [Online]. Available: <https://doi.org/10.1145/3520304.3528772>
- [8] —, "Dissecting copy/delete/replace/swap mutations: Insights from a gin case study," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. GECCO '22. New York, NY, USA: ACM, 2022, p. 1940–1945. [Online]. Available: <https://doi.org/10.1145/3520304.3533970>
- [9] V. Terragni, Y. Liu, and S. Cheung, "CSNIPPEX: automated synthesis of compilable code snippets from q&a sites," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, A. Zeller and A. Roychoudhury, Eds. New York, NY, USA: ACM, 2016, pp. 118–129. [Online]. Available: <https://doi.org/10.1145/2931037.2931058>
- [10] ESLint, "Find and fix problems in your javascript code - eslint - pluggable javascript linter," 2023. [Online]. Available: <https://eslint.org/>
- [11] Microsoft, "Typescript: Javascript with syntax for types," 2023. [Online]. Available: <https://www.typescriptlang.org/>
- [12] —, "Working with javascript in visual studio code," 2023. [Online]. Available: <https://code.visualstudio.com/docs/nodejs/working-with-javascript>
- [13] PMD, "Pmd," 2023. [Online]. Available: <https://pmd.github.io/>
- [14] R. Cottrell, R. J. Walker, and J. Denzinger, "Jigsaw: A tool for the small-scale reuse of source code," in *Proceedings of the International Conference on Software Engineering (ICSE)*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, p. 933–934. [Online]. Available: <https://doi.org/10.1145/1370175.1370194>
- [15] F. S. Ocariza, Jr, K. Pattabiraman, and A. Mesbah, "Vejovis: Suggesting fixes for javascript faults," in *Proceedings of the International Conference on Software Engineering (ICSE)*, ser. ICSE 2014. ACM, 2014, pp. 837–847.
- [16] M. Lajkó, V. Csuvik, and L. Vidács, "Towards javascript program repair with generative pre-trained transformer (gpt-2)," in *Proceedings of the International Workshop on Automated Program Repair*, ser. APR '22. ACM, 2022, pp. 61–68. [Online]. Available: <https://doi.org/10.1145/3524459.3527350>
- [17] GitHub, "Github copilot - your ai pair programmer," 2023. [Online]. Available: <https://github.com/features/copilot>
- [18] OpenAI, W. Zaremba, and G. Brockman, "Openai codex," 2021. [Online]. Available: <https://openai.com/blog/openai-codex>
- [19] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming *et al.*, "GitHub Copilot AI pair programmer: Asset or Liability?" *arXiv preprint arXiv:2206.15331*, 2022. [Online]. Available: <https://doi.org/10.48550/arXiv.2206.15331>
- [20] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, ser. MSR '22. New York, NY, USA: ACM, 2022, pp. 1–5. [Online]. Available: <https://doi.org/10.1145/3524842.3528470>
- [21] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. GECCO '12. New York, NY, USA: ACM, 2012, p. 959–966. [Online]. Available: <https://doi.org/10.1145/2330163.2330296>
- [22] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, "A journey among java neutral program variants," *Genetic Programming and Evolvable Machines*, vol. 20, no. 4, pp. 531–580, 2019. [Online]. Available: <https://doi.org/10.1007/s10710-019-09355-3>
- [23] J. Petke, B. Alexander, E. T. Barr, A. E. I. Brownlee, M. Wagner, and D. R. White, "A survey of genetic improvement search spaces," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. GECCO '19. New York, NY, USA: ACM, 2019, p. 1715–1721. [Online]. Available: <https://doi.org/10.1145/3319619.3326870>
- [24] D. Ginelli, M. Martinez, L. Mariani, and M. Monperrus, "A comprehensive study of code-removal patches in automated program repair," *Empirical Software Engineering (EMSE)*, vol. 27, no. 4, p. 97, 2022. [Online]. Available: <https://doi.org/10.1007/s10664-021-10100-7>
- [25] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: Genetic improvement research made easy," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. GECCO '19. New York, NY, USA: ACM, 2019, p. 985–993. [Online]. Available: <https://doi.org/10.1145/3321707.3321841>
- [26] Microsoft, "ts-fix," 2023. [Online]. Available: <https://github.com/microsoft/ts-fix>
- [27] T. M. T. Pham and J. Yang, "The secret life of commented-out source code," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, ser. ICPC '20. New York, NY, USA: ACM, 2020, pp. 308–318.
- [28] S. Baltes, L. Dumani, C. Treude, and S. Diehl, "Sotorrent: reconstructing and analyzing the evolution of stack overflow posts," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 319–330. [Online]. Available: <https://doi.org/10.1145/3196398.3196430>
- [29] S. Baltes and M. Wagner, "An annotated dataset of stack overflow post edits," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. GECCO '20. New York, NY, USA: ACM, 2020, p. 1923–1925. [Online]. Available: <https://doi.org/10.1145/3377929.3398108>