THE UNIVERSITY of ADELAIDE

Improving Developer Efficiency through Code Reuse

Brittany Reid

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The University of Adelaide

The School of Computer and Mathematical Sciences

Supervisors: Dr. Christoph Treude and A/Prf Markus Wagner

November 7, 2023

Contents

List of Figures

List of Tables

University of Adelaide

Abstract

Improving Developer Efficiency through Code Reuse

by Brittany Reid

Code reuse is an integral part of modern software development, where most software is built using existing software artefacts. Ranging from the copy-pasting of code fragments to the use of third-party libraries, developers frequently turn to the internet to find already-made solutions to difficult programming tasks and save development time. However, the large amount of libraries and code online can make finding the best solution difficult, and reuse is not necessarily straightforward. Most online code snippets do not run, meaning developers need to spend time correcting errors, and when example code snippets are meant to demonstrate API usage, this can present a barrier to using new libraries. This work studies ways to aid developers in the code reuse process, in order to improve their efficiency. We look at ways to more easily connect developers to the wealth of libraries and usage examples online from within their programming environment with our tool for Node.js, Node Code Query (NCQ). We then evaluate how well developers perform compared to the conventional code reuse process and found that developers using our tool solve tasks faster and have to try fewer libraries. Additionally, we study what problems online Node.js code snippets have and how to best correct them automatically, to save developers time in this step of the reuse process. We find that through the combination of the TypeScript compiler's error detection and codefixes, and our line deletion and custom fixes, we can increase the percentage error-free snippets in our dataset from 26.3% to 74.94%. Finally, we compare the emerging AI code snippet generation and pair programmer technologies to current online code snippet reuse practices, particularly looking at how snippets generated by GitHub's Copilot extension and those retrieved from Stack Overflow using Google might differ. We find that for the same set of queries, Copilot returned more snippets, with fewer errors and that were more relevant. Ultimately, this work provides further evidence of how automating the code reuse process can improve developer efficiency, and proposes a series of solutions to that end. Additionally, we provide a comparison between existing and emerging reuse processes. As the state of code reuse changes, helping developers understand the strengths of weaknesses of these approaches will become increasingly important.

Declaration of Authorship

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Brittany Reid

SEPTEMBER 2023

Acknowledgements

I would like to express my gratitude to my supervisors, Associate Professor Markus Wagner and Dr. Christoph Treude, for their support and guidance. Especially, I would like to thank both for always providing insightful feedback over the course of my research and thesis writing. I am also thankful to my co-authors, with whom it has been a wonderful experience to collaborate with.

I would also like to thank my friends and family for their encouragement. To my two closest friends, thank you for helping make sure that I didn't become a complete hermit over the last three and a half years.

Finally, I would like to thank the Australian Government Research Training Program for funding my studies, by which this research was made possible.

Chapter 1

Introduction

1.1 Background

Code reuse is the reuse of existing software artefacts to create new software. It constitutes a major aspect of modern software development, from copy-and-pasting small fragments of code ('code snippets') to reusing entire suites of existing classes and functions via software libraries. In 1968, Doug McIlroy proposed an idea of software development based on reusable, pre-existing software components (McIlroy et al., 1968), saying:

> When we undertake to write a compiler, we begin by saying 'What table mechanism shall we build?' Not, 'What mechanism shall we **use**?' but 'What mechanism shall we **build**?' I claim we have done enough of this to start taking such things off the shelf.

Now, over 50 years later, with the rise of the internet, large online registries of open-source software libraries such as NPM[1], Maven Central[2] and PyPI[3], and social platforms such as Stack Overflow[4], this mindset of 'use' over 'build' has become incredibly common. Over 79% of developers reported that they reused code from Stack Overflow, a popular programming question and answer site, in the last month (Baltes and Diehl, 2019). In fact, a significant portion of modern software development relies upon popular open-source libraries in each language that solve common problems; for example, software developers looking to test their code are unlikely to build their own test suite software, but to look at existing solutions such as JUnit for Java[5] and Mocha[6] or Chai[7] for Node.js. Popular frameworks like these are likely to be well-tested and actively maintained, and thus their usage allows development time to be directed elsewhere. This practice is ubiquitous in software development; one study analysing the frequency of code reuse in open-source projects found that 90% of analysed projects contained some form of code reuse (Heinemann et al., 2011). Repositories of open-source software libraries continue to grow, with NPM, the package

[1] https://www.npmjs.com/
[2] https://search.maven.org/
[3] https://pypi.org/
[4] https://stackoverflow.com/
[5] https://junit.org/junit5/
[6] https://mochajs.org/
[7] https://www.chaijs.com/

manager for Node.js, hosting over 1 Million libraries or 'packages', with use by industry companies such as Microsoft and Meta.

Motivations for reusing code vary; reuse saves development time and avoids having to 'reinvent the wheel', but it can also provide solutions when developers run into problems. Stack Overflow's 24 million questions and 35 million answers, which often include example code, demonstrates the ubiquity of this use-case (Stack Exchange, 2023). In general, code reuse is useful when developers may not have the time or knowledge to program something themselves. However, the process of reusing code itself is not simple; it can be time-consuming and introduce challenges of its own. Code sourced from online may be malicious, contain software vulnerabilities or bugs, or in some cases simply not run. The quality of online code also varies; only 8.41% of Java answers on SO containing compilable code (Terragni, Liu, and Cheung, 2016), and similarly, Yang, Hussain, and Lopes (2016) found that only 20.00% of JavaScript snippets could be run. This means that when developers copy-and-paste code from online, they may need to spend significant time making changes before they can run it. Additionally, online code may violate licence agreements; for example, multiple studies (Baltes, Kiefer, and Diehl, 2017; An et al., 2017; Ragkhitwetsagul et al., 2021) found instances where answers on Stack Overflow contained code copied from existing projects without providing attribution, and that many instances of reuse from Stack Overflow did not link back either, allowing stolen code to propagate freely online. This has an additional effect of enabling this code to become outdated: software updates may fix bugs or make changes to the API that are not carried over to unattributed code snippets. Without attribution, when a snippet no longer works, developers cannot easily trace the source to see these updates themselves either.

The complicated dependency chain of many software libraries also means that many projects rely on a number of additional libraries both directly, and indirectly, and developers may not be aware of all of them. On average, each package on NPM directly depends on 5.9 other packages (Kula et al., 2017). Crucial dependencies can simply disappear overnight; such as the case of NPM's `left-pad`, a tiny package that added characters at the beginning of a string. When the developer of `left-pad` deleted this library, many projects subsequentially broke because they or their dependencies directly or indirectly depended upon it. In some cases, developers may introduce malicious code into their libraries intentionally; in the case of NPM package `node-ipc`, as a form of protest against the war in Ukraine, this package was programmed to wipe the host system's data when it detected an IP address from Russia or Belarus (Kula and Treude, 2022). Instances like these show the amount of trust that is placed in reusable software artefacts, and how that trust can be exploited.

These risks highlight a real need for developers to take care when reusing code. However, developers often reuse code in situations where they do not have complete understanding; they often look to code reuse when they do not know how to do something, and thus, they may not completely understand the risks or how to mitigate them. The search space they navigate is also incredibly large and can make finding the 'best' library or code snippet for a specific purpose difficult and time-consuming. Nevertheless, code reuse has benefits for

efficiency, while difficult to quantify; for example, a return of investment of 400% has previously been observed (Lim, 1994).

1.2 Research Problem and Hypothesis

Though code reuse in its many forms is a common aspect of software development and benefits development greatly, it is not without risks, and processes involved can be time-consuming. The primary focus of this research thesis is in how the processes behind code reuse can be made more efficient, in order to improve developer efficiency when reusing code. This work both investigates methods for improving efficiency and, to that end, identifies challenges that developers experience in order to address them.

The typical manual process of code reuse (illustrated in Figure 1.1) is a three-step process from task to solution of 1) **search**; 2) **integration** and 3) **testing**. Developers identify a task they need to perform, or possibly a problem they need to solve. They formulate queries and search online for high quality code snippets or libraries. Then, they attempt to integrate the code or library within their existing project. This can mean making changes to code snippets for the new context, or installing and setting up new libraries. Finally, they check that the code works and is able to solve their problem. Challenges may arise at any of these stages, and developers may need to go back and retry; for example, switching between integration and testing, or evaluating multiple pieces of code, before they land on a suitable solution.

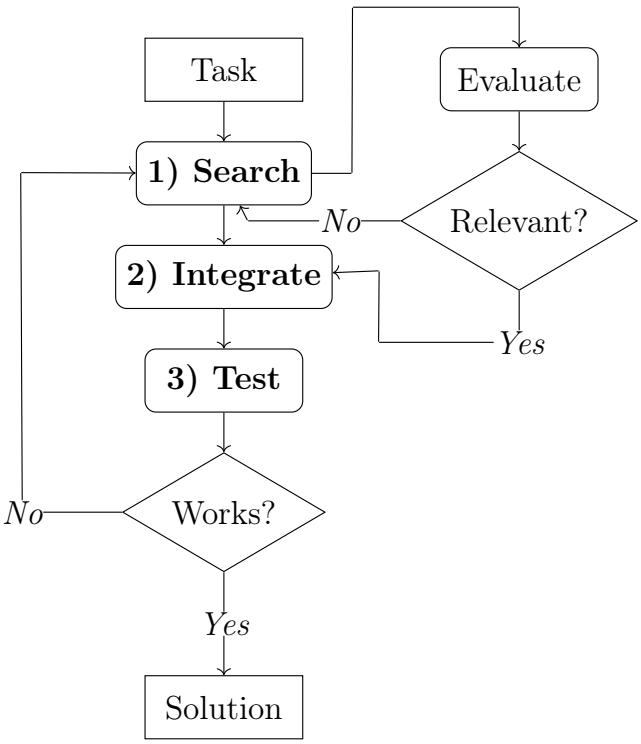

FIGURE 1.1. Code reuse process.

The purpose of automating code reuse is to make improvements at any of these steps in the manual code reuse process, in order to improve developer

efficiency. Automated code reuse systems may include integration of search tools within the programming environment to reduce context switching, the evaluation of large amounts of code to find the best snippet beyond manual capabilities, automated integration and error correction to reduce developer work involved and ways to make testing code functionality a smoother process. Additionally, the emergence of artificial intelligence code generation models such as OpenAI's Codex[8] threaten to change the landscape of code reuse, just as social media changed code reuse to a collaborative, online process (Storey et al., 2010).

Currently, code reuse predominately takes place online in the web browser, but this switching between the editor and web browser, a form of context switching, may negatively impact productivity. Code reuse involves multiple switches like this. First, developers encounter a situation programming they require a code reuse solution for must leave their programming to begin the search process in the browser. They need to formulate a query in natural language that describes the problem or task that takes place in the code-space. When they find a code snippet, they must return to the editor to integrate and test. If they encounter problems, they must return to the web browser and try again. Existing code reuse tools (Campbell and Treude, 2017; Ponzanelli, Bacchelli, and Lanza, 2013) integrate search within the editor to reduce context switching. The theory behind code recommender tools is that the way developers interact with their environments can be better adapted for this common programming task.

Because the ultimate goal of code reuse is to find the best snippet for a given task, automated systems excel at this problem. Where a human developer can only look at so many reuse candidates before choosing a 'good enough' solution (and research has shown that the majority of web search users do not go beyond the first page of results (Jansen, Spink, and Saracevic, 2000)), automated systems are capable of leveraging large datasets to evaluate and find this snippet quickly. If a perfect snippet doesn't exist, changes can be made automatically. This reduces the need for developers to spend time correcting erroneous online code; particularly, as code reuse tools may be of benefit to beginner developers who lack the domain knowledge to fix minor errors themselves. As part of evaluating code snippets, developers also test the functionality of code; a broader challenge of code reuse is how to evaluate the functionality of code.

Large language model (LLM) code generation (for example, OpenAI's Codex) functions as a form of code reuse as it 1) generates snippets that are then used similarly (copy-and-pasted, integrated and then tested) and 2) the output is generated from a corpus of existing code (a sort of reuse through synthesis). AI Tools such as GitHub's Copilot[9] (using Codex) operate within an IDE and take existing code context and cursor position as the input, and generate output for that context; the result is code snippets that may require less integration. These large language models can also be instructed with natural language to make changes or fix code. While it remains to be seen how useful these LLMs

[8] https://openai.com/blog/openai-codex
[9] https://github.com/features/copilot

are compared to current code reuse, and how may they shape future development habits, the functionality LLMs bring will make changes to the field of code reuse in some way.

The following sections provide a motivating example of the problems that occur during code reuse, through a typical code reuse scenario, then discusses the individual problems and solutions at each step.

1.2.1 Motivating Example

```
1 var path = "file.csv";
2 function main(){
3
4 }
5
6 main();
```

FIGURE 1.2. Example of a developer's Node.js file prior to reuse.

This example demonstrates a typical manual code reuse situation, wherein a developer would like to find a code snippet to solve a problem. Consider a case where a developer would like to read in a CSV file in Node.js. While this can be a relatively simple task, the developer is confident a solution already exists online that allows them to avoid writing their own code to parse CSV files. Additionally, if they're to use an existing library, it may be well-tested and cover more cases than their own code; this would be useful if they need to read in other files in the future. Their existing code is shown in Figure 1.2: they define an empty function `main` and the path of the CSV file. They then call the function.

FIGURE 1.3. SO answer for "Parsing a CSV file using NodeJS"

The developer begins by searching online. They use a general search engine, in this case Google, to search for the query 'read csv file Node.js'. Developers

often use general search engines like this to find code solutions, which then direct them to more specific websites. Upon searching, Google retrieves a number of websites. In this case, the developer selects one of the first results: the Stack Overflow question "Parsing a CSV file using NodeJS"[10]. They select the first answer with a code snippet, shown in Figure 1.3.

The developer can see that this answer has a vote count of 94, indicating that at least 94 people found it useful. The question has 18 other answers, so information like vote count can help developers make decisions on what answers to consider, and Stack Overflow additionally sorts answers by these votes. The Developer selects this answer because it includes a code example they can easily reuse; they avoid the accepted answer in this case as it recommends an outdated library and provides no usage example.

```
1  var path = "file.csv";
2  function main(){
3      var fs = require('fs');
4      var parse = require('csv-parse');
5      var csvData=[];
6      fs.createReadStream(req.file.path)
7          .pipe(parse({delimiter: ':'}))
8          .on('data', function(csvrow) {
9              console.log(csvrow);
10             //do something with csvrow
11             csvData.push(csvrow);
12         })
13         .on('end',function() {
14             //do something with csvData
15             console.log(csvData);
16         });
17 }
18
19 main()
```

FIGURE 1.4. The developer's file after they copy-and-paste the code snippet into their function.

The first step of integrating the code snippet is for the developer to copy and paste it into their existing file. Figure 1.4 shows what the developer's file looks like after they paste the snippet into their existing main function. However, in this state the code has errors that will prevent running; the developer needs to make changes to integrate it correctly within their existing code.

In this case, the developer's IDE, for example Visual Studio Code, highlights two errors in the code that will prevent running the code. The third-party library `csv-parse` is not installed, and the variable `req.file.path` is undefined. To make the snippet runnable, the developer first installs the library via the NPM package manager. However, the Stack Overflow answer provides no real information about the library; the developer would need to search on their own to find out how active the library is, who maintains it and what libraries it

[10] https://stackoverflow.com/questions/23080413/parsing-a-csv-file-using-nodejs

```
+ var fs = require('fs');
+ var parse = require('csv-parse');
+
var path = "file.csv";

function main(){
-    var fs = require('fs');
-    var parse = require('csv-parse');
     var csvData=[];
-     fs.createReadStream(req.file.path)
+     fs.createReadStream(path)
-        .pipe(parse({delimiter: ':'}))
+        .pipe(parse({delimiter: ','}))
         .on('data', function(csvrow) {
             console.log(csvrow);
             //do something with csvrow
             csvData.push(csvrow);
         })
         .on('end',function() {
             //do something with csvData
             console.log(csvData);
         });
}

main()
```

FIGURE 1.5. The developer's file after the snippet has been integrated.

```
var fs = require('fs');
- var parse = require('csv-parse');
+ var {parse} = require('csv-parse');

var path = "file.csv";
function main(){
     var csvData=[];
     fs.createReadStream(path)
         .pipe(parse({delimiter: ','}))
         .on('data', function(csvrow) {
             console.log(csvrow);
             csvData.push(csvrow);
         })
         .on('end',function() {
             console.log(csvData);
         });
}

main()
```

FIGURE 1.6. The developer's file after correction.

depends on. In this case, the developer decides to just use the recommended library, as the answer is highly voted. Next, they replace the path to the CSV file with their previously defined `path` variable. They make additional changes to the structure of the code, and the delimiter option. Figure 1.5 shows the integrated example.

However, the file still does not run. The code example has made a mistake; the `csv-parse` library does not export a function. To access the 'parse' function, the developer needs to change way the library has been imported in the require statement. This problem may have arisen because the example has not been updated in some time, and the library may have changed at some point. Regardless, the developer debugs the issue by observing that the value of the exported object has a method `parse` they need to access, then corrects the require statement as needed. The corrective changes are shown in Figure 1.6.

With all errors corrected, their file should now run, printing each row of the CSV file and then the final data in an array structure. However, the developer would like to test their code, and maintain a test case to ensure it remains working. They devise a simple test case using the testing framework `mocha`, which they might already be using in their project anyway. In order to test their code, they need to make some changes so that it can easily be given an input and return an output.

```
1  var fs = require('fs');
2  var { parse } = require('csv-parse');
3
4  - async function main(){
5  + async function readCSV(file){
6  +    return new Promise((resolve, reject)=>{
7           var csvData=[];
8           fs.createReadStream(file)
9               .pipe(parse({delimiter: ','}))
10              .on('data', function(csvrow) {
11 -                console.log(csvrow);
12                  csvData.push(csvrow);
13              })
14              .on('end',function() {
15 -                console.log(csvData);
16 +                resolve(csvData);
17              });
18 +    })
19 }
```

FIGURE 1.7. Changes to the code in order to enable testing.

Figure 1.7 shows the changes the developer makes to their `main` function, now `readCSV`. First, they change the function to async, and add an argument for the file path, so it can be used to read more than a single file. Next, they take the asynchronous stream logic and wrap it in a promise that will complete when the entire file has been read. This allows the function to be waited for using the Node.js `await` keyword, so the final object can be used after the reading process is complete. They no longer print for each row, and they no

```
it("should read a CSV file", async ()=>{
    var testPath = "test.csv";
    var testData = "Name,Job\n" +
    "Alice,Developer\n" +
    "Bob,Developer\n";
    fs.writeFileSync(testPath, testData);
    var data = await readFile(testPath);

    assert.strictEqual(data.length, 3); //has three rows
    assert.strictEqual(data[0][0], "Name"); //has column name
    assert.strictEqual(data[0][1], "Job"); //has column job
    assert.strictEqual(data[1][0], "Alice"); //has first name
})
```

FIGURE 1.8. The `mocha` test case for creating and reading a CSV file.

```
var fs = require('fs');
var { parse } = require('csv-parse');

async function readCSV(file){
    return new Promise((resolve, reject)=>{
        var csvData=[];
        fs.createReadStream(file)
            .pipe(parse({delimiter: ','}))
            .on('data', function(csvrow) {
                csvData.push(csvrow);
            })
            .on('end',function() {
                resolve(csvData);
            });
    })
}

var path = "file.csv";
var csvData = await readCSV(path)
console.log(csvData)
```

FIGURE 1.9. Developers final code.

longer print the final object `csvData`; instead, this object will be returned by the function when the promise completes.

Figure 1.8 shows the `mocha` test case that the developer creates to test the function. This test case creates a test CSV to read, then runs the function. After running, it performs four checks to ensure the data was correctly read. First, it checks that the number of rows is consistent with the test data. This length check will also fail if the function returns `undefined`. Second and third, they check the column headings are correct. Fourth, they ensure that the first value 'Alice' is correct. On run, the test case passes. Figure 1.9 shows the developer's final code after successful reuse.

1.2.2 Search and Evaluation

As seen in the section above, searching for code is the first step in reusing code. It also involves an aspect of evaluation; developers have a large amount of code snippets to select from and make choices before integrating and testing about which ones make sense to attempt use with. Primarily, this involves the relevancy of the snippet to their search query and if it may implement the required functionality, but developers can also evaluate the quality of snippets before trying them. In the above example, Stack Overflow vote count is used to help guide snippet and library choice.

Existing research has established that most developers use generic search engines like Google to find code and libraries (Umarji, Sim, and Lopes, 2008), as illustrated in Section 1.2.1. Developers formulate a query and retrieve a series of webpages as results, and the first step of the subsequent evaluation is to determine which one they will visit, discarding results that aren't relevant. If they are presented with more options, such as multiple answers on a Stack Overflow question, they also need to evaluate these as well. When finding a code example, developers need to read the code to determine if it's worth trying. Similarly, research has investigated the many features of third-party libraries that developers look at when choosing a library to use, including documentation, maintenance and popularity.

The primary issue of code search is that the search space is incredibly large: the entire internet. This means that it is impossible for a human to evaluate every option manually. Developers must also formulate a natural language query that may impact what results they see. In general, developers are selecting the first suitable result.

The aim of automating code search is to direct developers to high quality code as fast as possible. Automated code search can pull large amounts of code from many sources, evaluate this code automatically and present the best code snippet to a developer. Alternative methods of searching may also be useful; developers take problems in code and convert them into natural language, in order to find solutions in code. Search relies on matching queries to the documentation surrounding code snippets and not the code itself. Automated search may reduce this gap. Additionally, large language model-based tools like GitHub's Copilot generate snippets not just from natural language queries, but based on

the surrounding code, which might enable them to return higher quality and more relevant snippets.

1.2.3 Integration

Once developers have found a candidate piece of code for reuse, they need to integrate it within their existing project. This process may be time-consuming, and in some cases, unsuccessful, meaning developer may have to return to the search step.

Integrating third-party libraries involves installing the library and ensuring compatibility with existing libraries. Developers then begin using the library in their code; in many cases, they take examples from documentation to get them started. Many online code snippets, from library documentation and sites like Stack Overflow, do not run without changes. This means that developers must spend time fixing bugs, but also adapting the code snippets for their own purposes. In Section 1.2.1, we see that the developer needs to make changes to the example for the type of CSV files they will be reading in. Some changes are structural, like how the developer moved the require statements to the start of the file in Figure 1.6. Others involve replacing undefined or placeholder values with their own, such as the CSV file path in the example. The changes needed for code snippets are often context-dependent, so even code snippets that are otherwise correct may not work in certain contexts.

Automating this integration step aims to reduce the work developers need to do to get code snippets working in new contexts.

1.2.4 Testing

Once developers have code that can be run, they need to ensure it performs the required functionality they need from it. This involves the testing stage. The testing process can range from simply running the code to ensure it works, to writing test cases such as in the motivating example given. Developers must formulate test input and expected output. In the motivating example, the developer's input is a basic comma separated CSV, and their expected output is an array of three rows and two columns. Testing is the final stage of code reuse, as a snippet that passes all tests has been successfully reused.

In Section 1.2.1, we also observe how a developer changes their code to be more easily testable. When developers are writing code, they need to think about how the code will be used. In this case, the developer knew they might want to read in other CSV files as part of this project, so they devise their test case to test for this, and their code to handle this.

Automating the stage of testing aims to help developers determine the runnability and quality of code, which are often related concepts. Testing is a difficult aspect of programming in general, especially ensuring that all necessary cases are being tested for. In the motivating example, the developer does not test for edge cases, such as if there may be whitespace or commas in the fields of their CSV file. Despite the difficulty of developing test cases, testing reveals much about code that syntactic and static analysis may not.

1.3 Justifications for the Research

This research aims to automate the code reuse process, in order to improve developer efficiency, as well as the quality of code being reused. Code reuse is a common part of software development, but is time-consuming and can be augmented by automated techniques. Existing approaches fail to automate entire workflows. By automating the code reuse process, developers can spend less time reusing code and spend more time programming new features. Automated techniques can also integrate measures of code quality, to prevent vulnerabilities and bad practice. Developers are often reusing code when they experience problems or don't know how to do something, intuitively they are more likely to make mistakes in this situation.

1.4 Overview of Thesis Structure

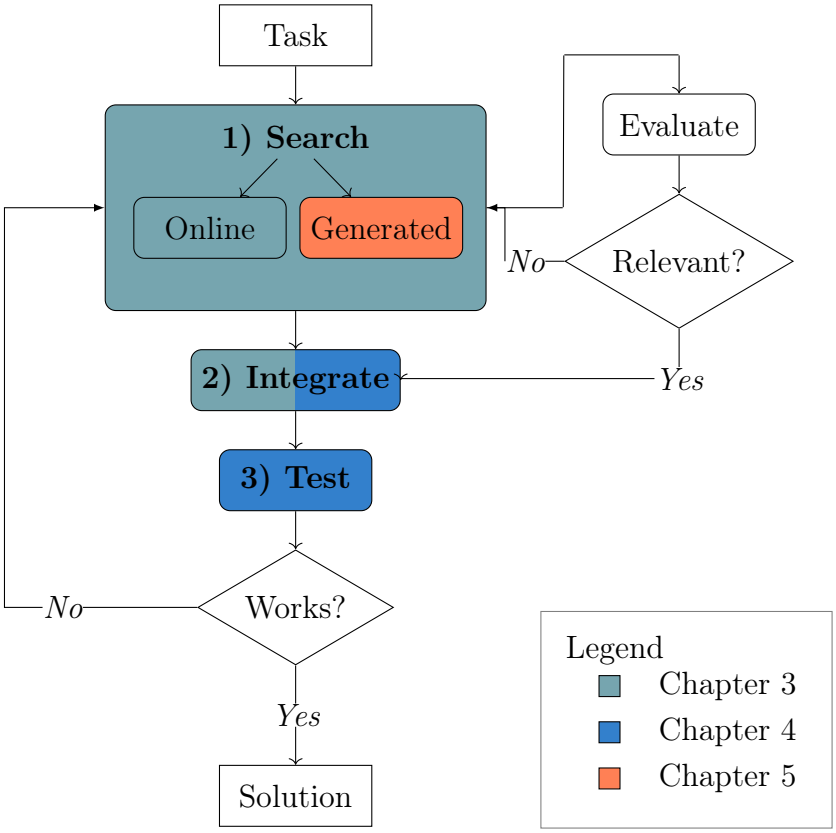

FIGURE 1.10. Modified Figure 1.1 showing the aspects of code reuse that each paper targets.

This section gives an overview of the thesis, which has been categorised into 6 chapters, as follows. First, this chapter introduces the problem and provides background. Next, Chapter 2 details the state of the existing literature in the area of code reuse. The subsequent chapters each detail a research paper that has been published during my candidature, for which I have contributed a significant amount of work and am the first author:

3. Connecting Developers to Libraries and Code Examples
4. Correcting Code Examples
5. Generating Code Examples

Finally, Chapter 6 summarizes the contributions of this thesis and discusses potential future work in the area of code reuse. The following sections detail each section and give a brief summary. Additionally, for each paper presented in this thesis, we contextualize its contribution in relation to the problem described in Section 1.1; in particular, which aspects of the code reuse process it focuses on automating. Figure 1.10, modified from Figure 1.1 highlights which parts of the code reuse process each paper addresses.

Chapter 2: Literature Review

Chapter 2 provides an overview of the topic of code reuse, through a literature review. We identify a series of topics related to code reuse: empirical studies on how developers reuse code, research into improving code snippet search, how developers select third-party libraries, improving library search, code synthesis as an alternative to search, the executability of online code and code correction. The existing state of literature is discussed, including gaps in the current work and where this research thesis fits in.

Chapter 3: Connecting Developers to Libraries and Code Examples

Brittany Reid, Marcelo d'Amorim, Markus Wagner, and Christoph Treude (2023b). "NCQ: Code Reuse Support for Node.js Developers". In: *IEEE Transactions on Software Engineering* 49.5, pp. 3205–3225. DOI: 10.1109/TSE.2023.3248113

Chapter 3 addresses the challenge of finding libraries and example code. As seen in Figure 1.10, this paper deals with both the search and integration steps of code reuse. The presented tool, Node Code Query (NCQ), helps connect developers to Node.js code snippets and libraries from the NPM registry within their editing environment. NCQ reduces the need for developers to context switch between editor and web browser, and connects developers to packages and snippets more quickly. Additionally, it corrects and sorts snippets by errors to aid developers with finding high quality snippets, thus reducing time spent correcting errors.

Code available on Github: https://github.com/Brittany-Reid/node_code_query

Chapter 4: Correcting Code Examples

Brittany Reid, Christoph Treude, and Markus Wagner (2023). "Using the TypeScript compiler to fix erroneous Node.js snippets". To appear in: *23rd IEEE International Working Conference on Source Code Analysis and Manipulation*

Chapter 4 addresses the challenge of correcting Node.js code snippets. As seen in Figure 1.10, this paper deals with both the integration and test aspects of code reuse. Using the TypeScript compiler, Node Code Correction (NCC) can detect and correct errors in Node.js code to aid developers when integrating incomplete code fragments. This work does not directly deal with testing, but it does show that analysing and manipulating code with the TypeScript compiler is viable in Node.js. Being able to generate Abstract Syntax Trees (ASTs) and determine type information is valuable for the ability to automate testing.

Code available on Github: `https://github.com/Brittany-Reid/node_code_correction`

Chapter 5: Generating Code Examples

Brittany Reid, Earl Barr, Markus Wagner, and Christoph Treude (2023a). *Copy-Paste vs. Code Generation: Contrasting Stack Overflow and GitHub Copilot.* Being revised for submission to IEEE Transactions on Software Engineering

With the rise of large language model code generation, developers may no longer need to look online for code snippets, but generate their own for a given context. Chapter 5 looks at the differences between current code synthesis via GitHub copilot (a LLM code generation embedded within the IDE), and code reuse via Stack Overflow and Google. As seen in Figure 1.10, this paper deals with the search aspect of code reuse; generated codes fill a similar purpose and 'reuse' in a less direct sense, where a model is trained on existing code to generate new snippets. However, developers still need to integrate and test this code. If generated code provides benefits over online code reuse, developers may look to it instead of online sources when they need snippets. In doing so, they may reduce the need for integration; in this work, we observed that Copilot snippets had fewer errors.

1.5 Related Literature

In addition to the above-mentioned papers, I have contributed to the following papers during the duration of candidature that are not related to the research topic of this thesis directly, or cannot be included:

8. Bodin Chinthanet, **Brittany Reid**, Christoph Treude, Markus Wagner, Raula Gaikovina Kula, Takashi Ishio, Kenichi Matsumoto. "What makes a good Node.js package? Investigating Users, Contributors, and Runnability." arXiv preprint arXiv:2106.12239. 2021.

9. **Brittany Reid**, Markus Wagner, Marcelo d'Amorim, and Christoph Treude. "Software Engineering User Study Recruitment on Prolific: An Experience Report." 1st International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES). 2022.

Additionally, this thesis cites the following work I have contributed to prior to my candidature:

10. **Brittany Reid**, Christoph Treude, and Markus Wagner (2020). "Optimising the Fit of Stack Overflow Code Snippets into Existing Code". In: GECCO, 1945–1953

    Code available on GitHub: https://github.com/Brittany-Reid/nlp2testablecode

Chapter 2

Literature Review

Because code reuse is a complicated and multistep process, research into automating code reuse focuses on many different areas. We look at studies on how developers currently reuse code, in order to understand the impact of reuse, what challenges they have and how the process can be improved. We are also concerned with the landscape of code and libraries that developers interact with online; that is, what types of snippets and libraries exist and how useable are they? Additionally, we look at code synthesis as an alternative for code reuse.

This literature review, adapted from the related work sections of the included publications, covers the following broad topics: the impact of code reuse in software development (Section 2.1), how to connect developers to code snippets and libraries (Section 2.2), how to help developers integrate code snippets into their existing projects (Section 2.3) and how to help developers test reused code (Section 2.4).

2.1 Empirical Studies on Code Reuse

Many studies have established the advantages and disadvantages of code reuse as a practice among software developers (Lim, 1994; Baltes and Diehl, 2019; Basili, Briand, and Melo, 1996; Mockus, 2007; Mohagheghi et al., 2004; Sojer and Henkel, 2010; Heinemann et al., 2011).

Lim (Lim, 1994) observed that code reuse improved code quality, increased productivity and reduced development time within two experiments at Hewlett-Packard (HP) focused on reusing already developed programming artefacts at the company. It was found that reuse contributed to accumulative bug fixes, leading to better quality code. Additionally, Lim identifies that, through reuse, expertise can be funnelled to creating new artefacts, that less experienced developers can then reuse. There is also an identified cost to code reuse, in that artefacts must be maintained. Though this work pertains to code reuse within an organization, the effect of accumulative bug fixes can also be seen in the collaborative nature of open source software online, and social platforms like Stack Overflow. *Reusers* of libraries may contribute back to the software, improving its quality, and in the case of Stack Overflow, users can edit answers and provide feedback via comments. Basili *et al.* found similar productivity gains and defect reduction from reuse in a study of students at the University of Maryland (Basili, Briand, and Melo, 1996).

Baltes and Diehl (Baltes and Diehl, 2019) surveyed 122 active Stack Overflow and GitHub users to find that 79% had "copied or adapted a code snippet from Stack Overflow" in the last month, and that half of respondents did so with no reference back to the Stack Overflow answer. Sojer and Henkel (Sojer and Henkel, 2010) surveyed 686 open source developers on their reuse practices, finding that experienced developers adopt reuse practices more often than non-experienced developers. They also observed that 30% of open source software projects included components from different projects. Mockus looked at code reuse at the file level in 38.7 thousand open source projects and found that approximately half of all files were used in at least one other project (Mockus, 2007). Another study (Heinemann et al., 2011) looked at 20 popular Java open source projects on SourceForge[1] and found that 90% contained code reuse in some form.

Our work on NCQ (Chapter 3) builds on the observations made in these previous studies: that there is a vast amount of code examples on the web today, that developers are already trying to use. We provide a platform for code reuse in Node.js where developers can experiment with different examples and libraries.

2.2 Search

Much work has focused on how to better connect developers to suitable code snippets and libraries, and what factors developers look at when selecting libraries to reuse. This section covers library selection, code search, and code synthesis as an alternative for code search.

2.2.1 Third-Party Library Selection

Developers looking to use a library in their projects often have a wealth of options to select from, with over 1 million packages hosted on the Node Package Manager (NPM) alone. For a specific functionality, there may be multiple options and forks of those options, and selecting the best package can be difficult. Research on what features developers look at when evaluating packages has identified factors such as documentation, performance, test coverage, security, maintenance, updates and the size of community as important to developers (Larios Vargas et al., 2020). However, much of the third-party library landscape consists of open source libraries, and there is a gap in the literature in how industry perceives and interacts with these libraries.

Some work has proposed tools to connect developers to libraries. El-Hajj and Nadi proposed LibComp (El-Hajj and Nadi, 2020), an IntelliJ plugin that assists the developer in selecting alternative libraries based on quality metrics associated with the libraries. LibComp does not perform code search, and it requires humans to provide the association of libraries to its knowledge base. Though existing search websites such as NPM's own site, and various websites for the Java Maven repository, often incorporate aspects of package quality,

[1] https://sourceforge.net/

popularity and maintenance into their search, our survey on Node.js developers for NCQ (Chapter 3) identified that many developers use general purpose search engines to find libraries. We aim to connect developers to high quality, relevant snippets by incorporating quality factors into our search. These measures may help connect developers to suitable packages faster.

2.2.2 Code Snippet Search

Code search has been extensively investigated (Campbell and Treude, 2017; Reid, Treude, and Wagner, 2020; Ponzanelli et al., 2014; Zhang et al., 2016; Brandt et al., 2010; Ponzanelli, Bacchelli, and Lanza, 2013; Gu, Zhang, and Kim, 2018; Xu, Vasilescu, and Neubig, 2022). Much of this work focuses on tools to facilitate finding snippets from within an IDE, such as NLP2Code (Campbell and Treude, 2017), Prompter (Ponzanelli et al., 2014), Blueprint (Brandt et al., 2010) and Seahawk (Ponzanelli, Bacchelli, and Lanza, 2013), in order to reduce context switching between the editor and browser. Additionally, this work can focus on different ways to formulate queries: as natural language or code itself, as in the case of AI assisted search approaches.

Brandt *et al.* (Brandt et al., 2010) proposed Blueprint, an in-editor tool to look for example code on the web. Experiments with Blueprint found that developers wrote higher quality code and search was significantly faster compared to regular web search. Similarly, Seahawk (Ponzanelli, Bacchelli, and Lanza, 2013) finds Stack Overflow code snippets in the editor, using the existing code context to automatically formulate search queries. Prompter (Ponzanelli et al., 2014) automatically notifies developers when it finds snippets on Stack Overflow that are similar to parts of the code under development. Campbell and Treude proposed NLP2Code (Campbell and Treude, 2017), an approach where developers use natural language queries to find code snippets. NLP2Code is integrated in the development environment to reduce context-switching, provides task suggestions and works entirely in the editor, with no additional views or windows. Reid *et al.* (Reid, Treude, and Wagner, 2020) proposed NLP2TestableCode, an improvement over NLP2Code that checks whether code snippets contain errors and attempts corrections before recommending them to the developer. Results indicate the ability to find compilable code snippets in 82.9% of cases, and "testable" code snippets (able to be converted into a function with input/output for testing) in 42.9% of cases. Our work, NCQ (Chapter 3), implements a similar code correction functionality, however, our code correction is limited by the lack of comprehensive error reporting in Node.js. We extend this work on error correction with NCC (Chapter 4),

There has been some work trying to improve code search via AI. When searching for code, developers often use natural language queries from which existing search systems extract keywords from, but there is a disconnect between code and natural language which makes connecting queries to code difficult. Gu *et al.* (Gu, Zhang, and Kim, 2018) proposed DeepCS, an approach that uses deep neural networks for code search. DeepCS uses a neural network model, dubbed CODEnn, embedding code snippets and natural language descriptions into a high-dimensional vector space. The intuition is that CODEnn

incorporates semantic information to the search in contrast to typical information retrieval approaches, which only use text similarity for search. Preliminary results showed that DeepCS outperforms alternative code search techniques. Xu *et al.* (Xu, Vasilescu, and Neubig, 2022) proposed NL2Code, an in-editor tool that combines code retrieval and code generation. Their tree-based semantic parsing model was trained on pairs of natural language and code mined from Stack Overflow and API documentation, and evaluation with developers found that perceptions of the tool were positive.

In summary, code search has been gaining traction in academia as researchers realise its ability to improve development productivity, specifically in cases involving the reuse of existing functionality over the creation of new functionality. However, automated code search tools still do not automate all steps of the code reuse process, including search, integration and testing. In this sense, the work presented in this thesis aims to do just that.

2.2.3 Code Synthesis

Generated code, for example from large language models (LLMs) such as OpenAI's Codex, fills a similar purpose as code reuse from online. There has been much work on code synthesis, the quality of outputted code, and more recently how developers might use these systems. Just as the social media revolution of code reuse (Storey et al., 2010) changed how developers reused code, from directly searching through source code for solutions to collaborative websites where other people have already found or written their own solutions, AI may change this once more. For example, Bird *et al.* (Bird et al., 2022) predicted that AI-powered tools will increasingly support developers, and that developer roles will evolve to focus more on assessing AI-generated suggestions. Where before a real human developer answers a Q&A question on Stack Overflow using their own knowledge base or code they find online, a LLM generates a snippet for a prompt from a database of training data, and does so on demand. In this way, LLM code generation functions as a type of code reuse because these models are trained on an existing corpus of code, then use this data to generate new snippets; e.g. they indirectly reuse code.

Typically, developers interface with these systems using a natural language query, and are given output in the form of one or more code snippets, that they then use and integrate into their own code. In this sense, the result is not fundamentally different from code snippets found online via web search. In the case of GitHub's Copilot, which embeds Codex within the editor as a plug-in for VSCode, snippets are generated using the existing code context and the cursor position; natural language queries can be embedded within the code as comments preceding the section to be generated. An example of this can be seen in Figure 2.1. It is this ability to generate custom snippets based on context, that could see developers needing to spend less time integrating snippets into their existing code.

Because copilot takes the existing code context as part of its input, it theoretically generates snippets that should fit within an existing project already. It follows then that these snippets should require no manual work to integrate.

```
import datetime

def parse_expenses(expenses_string):
    """Parse the list of expenses and return the
    list of triples (date, value, currency).
    Ignore lines starting with #.
    Parse the date using datetime.
    Example expenses_string:
        2016-01-02 -34.01 USD
        2016-01-03 2.59 DKK
        2016-01-03 -2.72 EUR
    """
```

FIGURE 2.1. Example input provided to copilot. The input is code with a natural language query embedded as a comment.

However, the reality of LLM generated code, for the moment, does not make this so. Existing work has established that generated code may contain errors and security issues, and LLMs have been observed to 'hallucinate' (Ji et al., 2023; Alkaissi and McFarlane, 2023); that is, to make up nonsense responses. The underlying systems behind AI mean that output can follow patterns that 'look' correct, but are actually incorrect on closer inspection. This pattern of behaviour has been called 'hallucinations'. More commonly, outside of programming contexts, such as the ChatGPT[2] tool, this manifests as confidently incorrect responses to a query, that would only be identified as incorrect by someone with domain knowledge. For programming, this presents an issue for novice developers who may not be able to identify if an LLM outputs harmful, malicious or vulnerable code.

Existing work has identified issues with the security of snippets outputted by Copilot (Pearce et al., 2022); approximately 40% of the generated programs were found to be vulnerable to high-risk cybersecurity weaknesses. However, Asare *et al.* (Asare, Nagappan, and Asokan, 2022) compared Copilot code to human code and found that it was better than humans at not introducing vulnerabilities; Copilot replicated the original vulnerable code about 33% of the time.

Some work has already looked at the performance of AI systems like Copilot. Sbania et al. (Sobania, Briesch, and Rothlauf, 2022) found similar performance between GitHub Copilot and genetic programming on program synthesis benchmarks, but genetic programming is less mature for practical use. Nguyen and Nadi (Nguyen and Nadi, 2022) assessed Copilot's code correctness and understandability using LeetCode questions and found that Java suggestions had the highest correctness, while JavaScript had the lowest. Dakhel et al. (Dakhel et al., 2022) studied Copilot's capabilities in fundamental algorithmic problems and human comparisons, concluding that Copilot can provide preliminary solutions but has limitations in advanced tasks. Yetistiren et al. (Yetistiren, Ozsoy,

[2] https://openai.com/blog/chatgpt

and Tuzun, 2022) systematically evaluated Copilot and reported a 91.5% success rate for generating valid code with varying correctness. Wong et al. (Wong, Kothig, and Lam, 2022) investigated Copilot-generated code quality using formal verification with Dafny and suggested that Copilot should not be solely responsible for programming. Erhabor (Erhabor, 2022) found that using Copilot can lead to code with significantly slower running times.

Other studies have explored the usability and potential impact of GitHub Copilot and other AI-powered programming assistants. Puryear and Spring (Puryear and Sprint, 2022) evaluated Copilot's impact on novice programmers, finding that it generated mostly unique code with human-graded scores ranging up to 95%. Peng et al. (Peng et al., 2023) found that using GitHub Copilot increased developers' task completion speed by more than 50%. On the other hand, Vaithilingam et al. (Vaithilingam, Zhang, and Glassman, 2022) conducted a user study and found that Copilot did not directly improve task completion time or success rate. Imai (Imai, 2022) compared Copilot to human pair programming and discovered that Copilot increased productivity but produced lower-quality code. Ziegler et al. (Ziegler et al., 2022) assessed Copilot's impact on productivity, finding that developers' perception of increased productivity depended on the acceptance rate of suggestions. Zhang et al. (Zhang et al., 2023) analyzed Copilot usage on Stack Overflow and GitHub Discussions and discovered that it was mostly used with JavaScript, Python, and Node.js, providing benefits and limitations. Jaworski and Piotrkowski (Jaworski and Piotrkowski, 2023) surveyed developers, revealing divided opinions and security concerns about Copilot. Additionally, the readability and robustness of generated code is an issue. Al Madi (Al Madi, 2022) accessed code readability and found that developers should be cautious of complacency and automation bias when working with model-generated code. Mastropaolo et al. (Mastropaolo et al., 2023) found that modifying natural language descriptions can result in different code recommendations in almost half of all cases, raising questions about the robustness of deep learning-based code generators.

While online code reuse must also contend with malicious, buggy or vulnerable code, and bad actors, LLM models generate code with no context. There is no external information that can be verified, such as who the author of a snippet or library is, or how commonly used the snippet is. Sites like Stack Overflow involve collaborative aspects such as voting, enabling higher-quality snippets to rise above lower-quality or outdated ones. In a way, the lack of collaborative 'rating' for solutions may be a negative. Currently, SO bans AI answers from its website (Makyen, 2023). What is currently lacking in this area of research, is a comparison between existing reuse and generated code. In Chapter 5, this is what we aim to do, looking at the output of both Copilot and Stack Pilot.

2.3 Integration

There has been some work on aiding developers in integrating code snippets from online into their projects. Primarily, this focuses on helping developers resolve errors that arise from copy-paste reuse. However, in order to fix errors in reused code, we require methods to automatically identify these errors. Of

main concern is the observation that most online code snippets do not run; this means that related areas, such as automated program repair and bug detection techniques that rely on executable code, cannot be applied. Much of the area of code fragments thus deals with static analysis (which does not require executability), and how to make snippets into some runnable form. This section looks at the state of code snippets (what problems they might have and how to identify them) and methods for automated code correction.

2.3.1 Code Executability

Existing work has established that the majority of online code snippets do not run (Chinthanet et al., 2021; Yang, Hussain, and Lopes, 2016; Mirhosseini and Parnin, 2020; Horton and Parnin, 2018), which can present a challenge to developers attempting to reuse these code snippets. In one study of online coding tutorials (Mirhosseini and Parnin, 2020), it was found that only 26% of the included snippets were able executable, which may constitute a problem especially when developers turn to such tutorials for topics they are inexperienced with. An evaluation of GitHub Gists found similar results: 25% of code snippets were runnable by default (Horton and Parnin, 2018). For Stack Overflow in particular, it has been observed that only 8.41% of Java answers on Stack Overflow contained compilable code (Terragni, Liu, and Cheung, 2016). In a study of code examples from NPM documentation, only 15.20% of ran successfully (Chinthanet et al., 2021).

Some systems have been proposed to help developers write and maintain executable code examples. Mirhosseini and Parnin (Mirhosseini and Parnin, 2020) proposed a system to annotate documents and encouraged the use of notebooks (e.g., Jupyter Notebooks (*Jupyter* 2020)) to improve executability of tutorials. Similarly, Melo *et al.* (Melo, Wiese, and d'Amorim, 2019) proposed Frisk, a system that enables QA forum users to describe problems (e.g., configuration problems) and share these problems with the community (e.g., Stack Overflow users), who could help fix the problems. Frisk uses Docker to enable developers consistently reproduce problems and collaboratively propose fixes.

However, none of this work tackles methods to automatically improve executability of code that already exists online. Instead, they rely on some manual effort in creating correct snippets to begin with. The primary issue of code snippets is that they are often created for example purposes and thus lack aspects that may be obvious from context. For example, a code snippet in a Stack Overflow answer may omit code that has already been provided in the question.

2.3.2 Static Analysis

Previous work has looked at error detection in JavaScript and Node.js, using parsers (Yang, Hussain, and Lopes, 2016), linters (Campos et al., 2019; Reid et al., 2023b) or runtime errors (Chinthanet et al., 2021; Yang, Hussain, and Lopes, 2016). 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 (Chinthanet et al., 2021;

Yang, Hussain, and Lopes, 2016). 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 (Reid et al., 2023b) and NLP2TestableCode (Reid, Treude, and Wagner, 2020) 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' (Campos et al., 2019).

Similar work in Java has made use of static analysis tools like PMD (PMD, 2023) and compilers to detect and correct errors in code snippets (Reid, Treude, and Wagner, 2020; Terragni, Liu, and Cheung, 2016; Licorish and Wagner, 2022a; Licorish and Wagner, 2022b). 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.

2.3.3 Automatic Code Correction

Prior work has attempted to improve the quality of code snippets sourced from online, which often have errors due to the nature of example code. Code snippets are often not intended to run, but to demonstrate functionality; for example, previous work has found that only 15.2% of code snippets in NPM documentation are able to run (Chinthanet et al., 2021). However, previous work like NLP2TestableCode (Reid, Treude, and Wagner, 2020) or CSnippEx (Terragni, Liu, and Cheung, 2016) has focused on Java code snippets, and employed the use of the Eclipse Compiler to detect and fix errors. CSnippEx implements Eclipse's built-in Quick Fixes, while NLP2TestableCode employs a set of heuristic fixes and a line deletion algorithm to correct snippets. However, dynamic languages like JavaScript and Python do not have compilers, and work in these languages has previously made use of linters and parsers to detect errors (Campos et al., 2019; Yang, Hussain, and Lopes, 2016). To our knowledge, no work has attempted to combine this error detection with code correction as we do in NCQ.

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 (Terragni, Liu, and Cheung, 2016), for Java, employs an existing suite of fixes from Eclipse, while NCQ (Reid et al., 2023b) does the same in Node.js using ESLint's fixes. NLP2TestableCode (Reid, Treude, and Wagner, 2020) in Java uses a set of custom heuristic fixes, and is able to insert missing import statements and variable definitions. Jigsaw (Cottrell, Walker, and Denzinger, 2008), 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 (Ocariza, Pattabiraman, and Mesbah, 2014) 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. (Lajkó, Csuvik, and Vidács, 2022) 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 (GitHub, 2023), a plug-in for VSCode that uses OpenAI's more advanced GPT-3-based Codex (OpenAI, Zaremba, and Brockman, 2021), 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 (Dakhel et al., 2022; Nguyen and Nadi, 2022). 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, as we test out in Chapter 4.

2.3.4 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 (Le Goues, Weimer, and Forrest, 2012; Harrand et al., 2019; Licorish and Wagner, 2022b; Licorish and Wagner, 2022a; Petke et al., 2019; Ginelli et al., 2022). 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 (Reid, Treude, and Wagner, 2020), 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 (Reid et al., 2023b) implements a similar functionality for Node.js, using ESLint to evaluate errors. Similarly, Licorish and Wagner (Licorish and Wagner, 2022a) combined static analysis with the Gin genetic improvement framework (Brownlee et al., 2019) (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.

2.4 Testing

Developing good test cases is difficult, and thus there has been much work on ways to automate software testing and help developers generate test cases for projects (Hanford, 1970; Bird and Munoz, 1983; Duran and Ntafos, 1984; Pacheco et al., 2007; Tonella, 2004; Fraser and Arcuri, 2011). However, in the code reuse context, testing is more concerned with ensuring the runnability and functionality of candidate code. Primarily, developers are concerned with if a snippet or library runs, and then if so, it accomplishes some task they need. For example, testing as part of the reuse process may simply involve executing the code and checking if the output is correct manually. In this sense, automating code reuse looks at how to better integrate testing into the reuse process. Additionally, existing software testing techniques, such as testing frameworks, runners, and generators can still be helpful.

In Section 1.2.1 we describe a manual reuse situation where a developer first executes their code, then writes a function and `mocha` test case. To automate this process, we need to be able to make candidate code into a 'testable' form, and to automatically generate test cases, with as little developer input as possible. The intended aim of automated testing in a reuse context is to integrate runnability as a measure of the quality of code candidates.

There has been much work on ways to automatically generate test cases. The area of randomized test case generation is a type of black box testing using random inputs (Hanford, 1970; Bird and Munoz, 1983; Duran and Ntafos, 1984; Pacheco et al., 2007; Tonella, 2004; Tufano et al., 2022; Wang et al., 2023); that is, the code itself under test is not used to generate test cases. Tonella (2004) proposed evolutionary test generation for classes (eToc), which uses genetic algorithms to generate and mutate test cases using coverage as a measure of fitness. Class tests often involve first constructing a class, then running a series of class methods; the available mutation operations are thus to alter the constructor, add or remove method calls, or change input values. EvoSuite (Fraser and Arcuri, 2011) used a similar evolutionary approach, however, it also improved upon existing approaches by optimizing entire test suites, not just single test cases, for coverage.

More recent work has looked at various AI techniques for automating the testing process. Tufano et al. (2022) propose a transformer-based approach to generate assert statements for test cases from functions. Wang et al. (2023)

investigated how large language models may aid software testing and test case generation; specifically, because of their ability to solve natural language processing tasks. In the context of code reuse, developers typically have a natural language task that specifies code function, and may be useful for test case generation using LLMs. Within the area of automated testing, there is the matter of how to determine what test output is correct, or the *oracle*. Test cases that crash a program or never complete can be detected, but other expected results are harder to predict. A common approach, as discussed above, is to use code coverage to generate test cases and suites, but generated test cases still require manual verification. In this sense, the ability for LLMs to understand a developer's natural language task may aid in determining expected output.

Some techniques have already been proposed to use test cases to help developers find runnable snippets. NLP2TestableCode (Reid, Treude, and Wagner, 2020), a plug-in for the Eclipse IDE, integrates testing into the editor, automatically correcting and then testing snippets in an attempt to find snippets that run without a developer needing to test each one. To do this, NLP2TestableCode converts snippets into functions with input/output variables and generates JUnit test cases based on type information. To address the disconnect between search queries and code, test-driven code search has also been proposed as an alternative to natural language queries; CodeGenie (Lazzarini Lemos, Bajracharya, and Ossher, 2007) helps connect developers to code snippets by using test cases to describe what functionality they want.

2.5 Conclusion

In summary, code reuse is a wide field, where multiple techniques from different areas may be relevant in automating the search, integration and testing stages of reuse. The following chapters thus each detail work that builds from this literature in order to work towards automating code reuse.

Chapter 3

Connecting Developers to Libraries and Code Examples

Related Publication: This chapter is based on our paper: Brittany Reid, Marcelo d'Amorim, Markus Wagner, and Christoph Treude (2023b). "NCQ: Code Reuse Support for Node.js Developers". In: *IEEE Transactions on Software Engineering* 49.5, pp. 3205–3225. DOI: 10.1109/TSE.2023.3248113 (Reid et al., 2023b).

In the literature review (Chapter 2), we identify how code reuse is an important part of software development, and that the adoption of code reuse practices is especially common among Node.js developers, where the Node.js package manager, NPM, indexes over 1 Million packages. Developers then often seek out packages to solve programming tasks. However, due to the vast number of packages, selecting the right package is difficult and time-consuming. With the goal of improving productivity of developers that heavily reuse code through third-party packages, we present *Node Code Query* (NCQ), a Read-Eval-Print-Loop environment that allows developers to 1) search for NPM packages using natural language queries, 2) search for code snippets related to those packages, 3) automatically correct errors in these code snippets, 4) quickly setup new environments for testing those snippets, and 5) transition between search and editing modes. In two user studies with a total of 20 participants, we find that participants begin programming faster and conclude tasks faster with NCQ than with baseline approaches, and that they like, among other features, the search for code snippets and packages. Our results suggest that NCQ makes Node.js developers more efficient in reusing code.

Code available on Github: https://github.com/Brittany-Reid/node_code_query

3.1 Introduction

Node.js is a popular JavaScript runtime (W3Techs, 2020) often used to develop server-side applications. The Node.js package manager, NPM, hosts over 1 million packages (Tal and Maple, 2020), with a typical package recursively depending on several others; on average each package directly depends on 5.9 other packages (Kula et al., 2017). Despite such a rich package ecosystem, finding the right package and figuring out how to use it can be time-consuming. The vast number of options means that developers are unable to gain detailed understanding of all potential selections and thus make decisions based on limited data. The typical package search process is similar to the code reuse example explained in Section 1.2.1, where developers search online (*e.g.* using a general purpose search engine or the NPM website), then evaluate relevant packages by 1) installing the selected package and 2) testing the functionality of that package (*i.e.*, running code snippets) to decide if the package is fit for purpose. If a developer needs to try multiple packages (for example, due to a lack of features), this process can become time intensive.

We conducted a preliminary study to understand the challenges of developers in finding packages. Our survey of 55 Node.js developers revealed that package use is indeed a common part of Node.js development, yet most developers encountered challenges in finding packages, with the most common challenge being insufficient documentation and code examples. To sum up, developers have challenges determining *how to use* the packages they find.

To address the challenges of package search in the Node.js ecosystem, we propose Node Code Query (NCQ), a custom Read-Eval-Print-Loop (REPL) (Sandewall, 1978) which integrates 1) NPM package search using natural language queries, 2) code snippet search, 3) automatic code snippet error correction, 4) automated setup of environments for testing those snippets, and 5) the ability to transition between search and editing modes. We conjecture that NCQ is a useful development aid in an environment where much coding is related to connecting snippets of code as opposed to developing code from scratch. To reduce the effect of context switching on performance, NCQ integrates multiple code reuse steps into one tool; package and code search, a code editor and a shell to execute code and install packages.

Existing ideas from the literature in code and library search (Campbell and Treude, 2017; Ponzanelli et al., 2014; Zhang et al., 2016; Brandt et al., 2010; Ponzanelli, Bacchelli, and Lanza, 2013; El-Hajj and Nadi, 2020; Reid, Treude, and Wagner, 2020) inspired NCQ. The novelty of NCQ is grounded on the application of these ideas in a different context. Additionally, existing work in automated code corrections for online code snippets (which uses static analysis to detect errors as most snippets do not run and do not have test cases) has focused on Java using the compiler(Reid, Treude, and Wagner, 2020; Terragni, Liu, and Cheung, 2016; Licorish and Wagner, 2022a). However, no work has focused on the application of similar fixes in Node.js, or more broadly JavaScript, which lacks such an error detection method. We conjecture that NCQ's interactive REPL can better assist developers in the common scenarios where they wish to try out new software libraries quickly in an isolated environment.

Developers can avoid making changes in their local programming environment when they want to borrow functionality from existing libraries. The combination of REPL, package and code search and code correction is non-trivial; they have been adapted for this new context to be intuitive within a command-line interface, and to reduce friction and context-switching. Our user studies aim to validate these assumptions.

To validate the usefulness of NCQ, we evaluated the tool with Node.js developers. Participants' activities were compared against a baseline comprising of internet access and Visual Studio Code, which we identified as the most popular editor for Node.js developers. We asked participants to complete two basic programming tasks, first using the baseline, then using only the tool. Our evaluation shows that participants were able to complete all tasks using NCQ without context switching, at least as quickly as when they used a well-established editor and had access to online search. Participant feedback for the tool was generally positive in regards to helpfulness, confidence in solutions and tool features. These results suggest that NCQ is a promising direction to fill an existing gap in tool support for Node.js code reuse, reducing the burden of context switching between searching and editing tools.

We make the following contributions:

- ⋆ **Approach.** A command line tool, NCQ that integrates documentation from the NPM database, within a REPL with the ability to create temporary environments, search for packages, install those packages, search for and run related code snippets and edit submitted code like a traditional code editor, along with an evaluation in two user studies with a total of 20 participants.
- ⋆ **Artifacts.** NCQ is publicly available on GitHub at `https://github.com/Brittany-Reid/node_code_query`.

3.2 Illustrative Example

In Section 1.2.1 we demonstrate the typical code reuse process. In Node.js this process is similar, but typically focuses on the use of NPM packages. In this section, we first describe the typical Node.js code reuse process, then we describe how NCQ could improve on this process. Let us consider a scenario where a developer would like to create a CSV file in Node.js. The developer would like to use a package to solve this task; this can be a simple task to manually code, but a package would save time and avoid mistakes. The following sections describe the workflow two developers followed when solving Task 3 of our user study. In the following, we describe the problem-solving process followed by two participants. One participant was asked to use the internet to find packages and to code in Visual Studio Code, while the other participant used only NCQ. Section 3.7.1 describes this task in more detail.

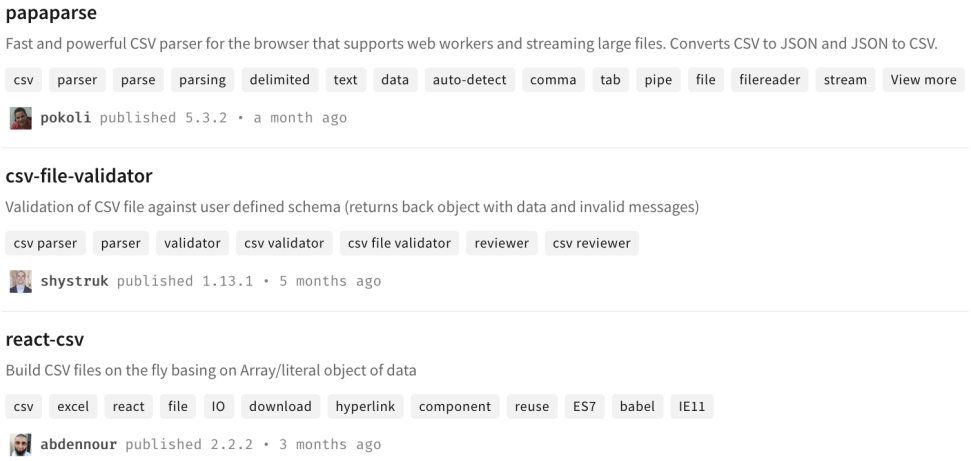

FIGURE 3.1. NPM website search results for "csv file".

3.2.1 Typical problem solving in Node.js

This section describes the workflow of one participant from our user study, using the internet and a traditional editor. We consider this workflow to be a good example of typical problem solving in Node.js.

The developer starts by using a general purpose search engine (all participants in our user study (Section 3.10) started with a general purpose search engine). In this case, they enter the query "!npm csv file" into search engine DuckDuckGo, which redirects them immediately to NPM's search for the query "csv file". Figure 3.1 shows the first three results for this query, however, none pertain to *creating* CSV files. After reading the names and descriptions of these packages, the developer refines their search, using the query "csv file create". The developer looks through the results for this query, finding the package `node-create-csv` which mentions converting an array into a CSV file, and selects this package.

After being taken to the README file for this package, the developer can see that it has some usage examples, including the example shown in Figure 3.2 which writes some data to a file.

```
1 const ObjectsToCsv = require('objects-to-csv');
2
3 const data = [
4   {code: 'CA', name: 'California'},
5   {code: 'TX', name: 'Texas'},
6   {code: 'NY', name: 'New York'},
7 ];
8 (async () => {const csv = new ObjectsToCsv(data);
9   await csv.toDisk('./test.csv');
10  console.log(await csv.toString());})();
```

FIGURE 3.2. Usage example from the README for `node-create-csv`. Note the incorrect package name.

The developer decides to try to use this package, so they create a new Node.js project. First, they make a new directory for the project, then in that directory,

they use the command `npm init -y` to initialise the project, which creates a special `package.json` file that stores metadata including a list of dependencies. With the project configured, the developer now installs the package with the `npm install node-create-csv` command.

With the package installed, the developer copies and pastes the example into their editor. They modify the data being written to match the task's requirements, however, when they attempt to run the code, Node.js reports the `MODULE_NOT_FOUND` error despite the package being installed. After some investigation, the developer realises that the example imports a different package, `objects-to-csv`, and the error refers to *this* package not being installed. The developer looks over the rest of the examples and sees the same issue, so considers this package to be buggy and moves on.

However, during our post-session investigation, we found that `node-create-csv` is a copy of an existing package with its documentation unchanged. Had the developer known, they could have changed the import or installed the named package instead to bug-solve, but developers often consider package documentation and examples important in evaluating package quality. The developer uninstalls the package using the command `npm uninstall node-create-csv` and decides to look at the other packages NPM returned, which may have working examples.

```
let converter = require('json-2-csv');
+ const fs = require('fs');

- let documents = [{Make: 'Nissan', ... },
+ let documents = [{Name: 'Alice',
+       Institution: 'Foo', Job: 'IT Manager' },
-   {Make: 'BMW', ...}];
+   {Name: 'Bob', Institution: 'Bar',
+       Job: 'Developer'}];
let json2csvCallback = function (err, csv) {
    if (err) throw err;
    console.log(csv);
+   fs.writeFile('./people.csv',csv,function(err) {
+       if(err) return console.log(err) })
};

converter.json2csv(documents, json2csvCallback);
```

FIGURE 3.3. The modifications the developer made to complete the task, with new lines marked in green with "+" and removed lines in red with "-".

The developer evaluates another three packages from their search results, reading their documentation, however none have any examples in their README files. Finally, the developer checks the `json-2-csv` package. In the README for this package, there are no usage examples, but a link to some examples on GitHub, where they find one example for writing a CSV file. They install the package and paste the example in their editor.

Figure 3.3 shows how the example code for this package was adapted to solve the assigned task. The original code snippet takes an array of data, `documents`,

and defines a callback function where the argument `csv` is printed to the console. The developer modifies it so the array contains their data, and in the callback, the string is written to a file called "people.csv". They run this code and the file is created successfully. In total, this participant took 18 minutes to complete this task.

To sum up, this approach can be time-consuming for the following reasons:

- Developers need to search different sources to locate potentially useful packages. When finding those package, developers need to read documentation and check examples, when available;
- Not all packages have examples, making evaluating package quality more difficult.
- The package of interest may *not* work as expected. Running code snippets is important to make a decision on the package to use. For that, the user needs to create temporary environments and (un)install packages on top of those environments to test different packages.

3.2.2 Problem solving in NCQ

This section illustrates how developers solve tasks using NCQ, a tool to help developers search for Node.js packages and experiment with related examples. We describe the workflow of another developer in our user study. The developer was asked to solve the same task from Section 3.2.1.

First, the developer starts the tool and uses the `repl` command to create a new Read-Eval-Print-Loop (REPL) instance, where they can install packages and execute code. The developer then uses the `.packages <search string>` command from within the REPL to search for packages using the search string. This command takes the developer to a list of packages and their descriptions, which they can scroll through. Figure 3.4 shows the output the REPL produces for the command `.packages csv`.

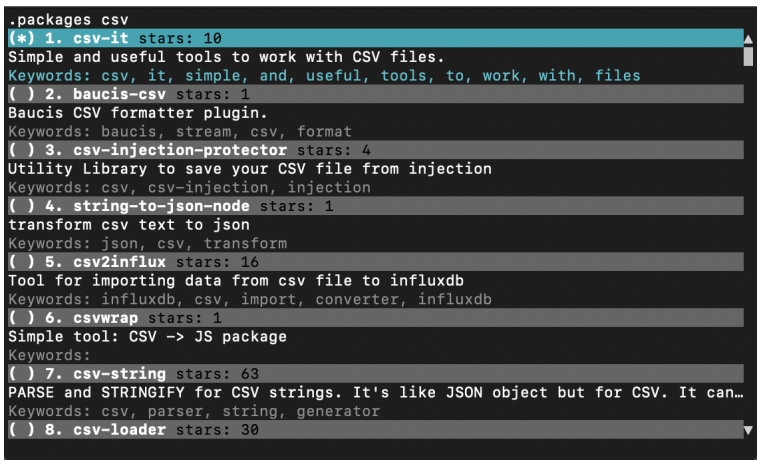

FIGURE 3.4. Results for the query '.packages csv'.

The list of packages allows for skimming many descriptions to quickly disregard irrelevant packages, for example, `csv2influx`, which appears to import

CSV files into a database. NCQ sorts packages based on their runnability, and excludes packages with no stars or no example code in their READMEs. The first entry, `csv-it`, looks promising so they select it with the enter key. Upon selection, the tool asks if they would like to install the package, to which they respond "yes".

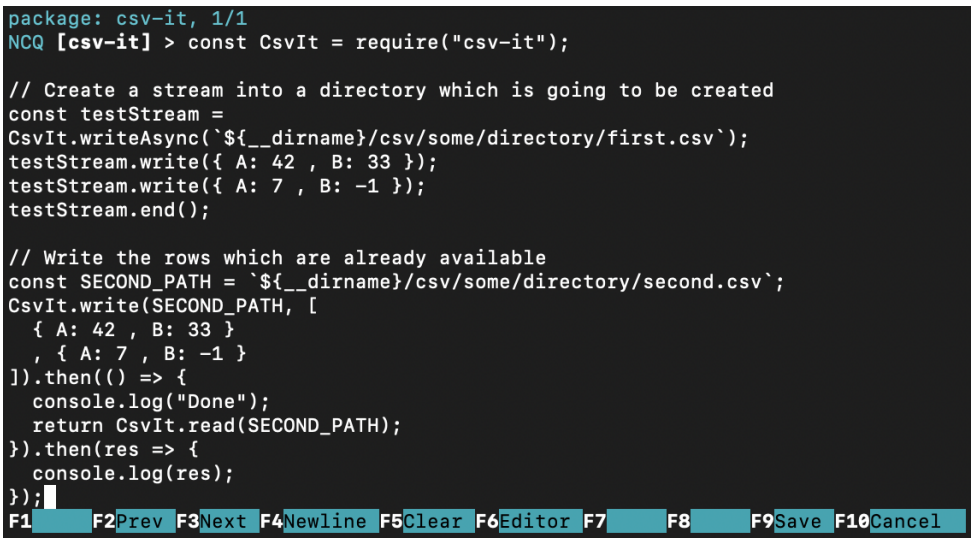

FIGURE 3.5. Cycling through code snippets for the package `csv-it` using the command `.samples`.

To evaluate a package, developers can use the `.samples <package name>` command to retrieve code snippets for a given package, extracted from NPM by scrapping the README file associated with a package. The developer uses the command `.samples` to see examples for the installed packages. Figure 3.5 shows the code snippet for `csv-it`. The code snippet autofills the command prompt; from here the developer can execute the code snippet like any other code in the REPL by pressing enter, or cycle through other code snippets by using the function keys F2 and F3. Each code snippet is evaluated using ESLint to correct errors and run through NCQ's error correction process. This process fixes errors, enforces consistent style rules on snippets and comments out error-causing lines, before being sorted by final error count to show error-free snippets first.

In this case, there is only one code snippet, which demonstrates how to write some data to a CSV file in two different ways. NCQ's automated fixes have added semi-colons where they are missing; Figure 3.6 shows the changes the developer makes to the code snippet to complete their task. First, the developer removes the second example. Next, they decide to run the code snippet by pressing enter and see that it works as expected; a file named "first.csv" is created with the columns "A" and "B" and two rows of values. After executing, the developer uses the `.editor` command, which allows them to edit their previously run code in a traditional editor, where they change the file name and data before saving. On save, the REPL re-runs the code successfully. In total, this participant took 15 minutes to complete the task.

```
const CsvIt = require("csv-it");

- const testStream = CsvIt.writeAsync(‘first.csv‘);
+ const testStream = CsvIt.writeAsync(‘people.csv‘);
- testStream.write({ A: 42 , B: 33 });
+ testStream.write({ Name: "Alice",
+   Institution: "Foo",  Job: "IT Manager"});
- testStream.write({ A: 7 , B: -1 });
+ testStream.write({ Name: "Bob",
+   Institution: "Bar", Job: "Developer"});
testStream.end();

- const SECOND_PATH = ‘second.csv‘;
- CsvIt.write(SECOND_PATH, [{ A: 42 , B: 33 } ,
-   { A: 7 , B: -1 }]).then(() => {
-  console.log("Done");
-  return CsvIt.read(SECOND_PATH);
- }).then(res => {console.log(res);});
```

FIGURE 3.6. How the code snippet in Figure 3.5 was adapted.

In summary, NCQ combines ideas from existing work in code and package search to aid code reuse by:

- Automatically setting up temporary environments that developers can use to experiment with packages;
- Allowing developers to search for packages without leaving their programming environment (El-Hajj and Nadi, 2020); developers only see packages with example code, making the package to code process faster;
- Sorting packages by runnability to prioritise functioning packages; the majority of developers in our user study were able to solve their task with the first package they found;
- Embedding code snippets within the editor (Campbell and Treude, 2017) and enabling developers to quickly try out different packages without leaving the tool, while providing immediate error feedback when code is submitted, via the use of the REPL design. Developers are easily able to check if code snippets work before making changes;
- The addition of a traditional editor mode so changes can be made to the REPL's state, which makes the REPL more intuitive for users;
- Automatically fixing errors and sorting snippets by number of errors to reduce the amount of work developers do to reuse code (Reid, Treude, and Wagner, 2020).

3.3 Dataset

NCQ uses a database of packages, package data and code snippets to enable its search. Like previous work (Reid, Treude, and Wagner, 2020), we implement

this database offline to avoid internet bandwidth issues. The following sections describe how package data and code snippets were mined. Table 3.1 shows the package data breakdown; our final dataset consists of 620,221 packages with 2,161,911 code snippets.

Packages as of May 2021	1,607,057
Packages with READMEs	1,298,170
Packages with non-empty READMEs	1,297,678
Packages with Node.js code snippets	620,221
Node.js code snippets	2,161,911

TABLE 3.1. Summary of dataset statistics.

3.3.1 Package Data

To enable the package and code search, we mined the NPM registry for NPM package data. We chose to restrict our focus to a single package manager to simplify automation. Many packages are also hosted across multiple package managers, so focusing on a single one limits data duplication and the size of our database. We selected NPM due to it being one of the most popular package managers for Node.js. The registry maintains a list of all package names at the URL `https://replicate.npmjs.com/_all_docs`. For each package, a JSON file containing package data (*e.g.*, README, description, author and link to the repository) can be found at the URL `https://registry.npmjs.org/<package>`.

There were 1,607,057 packages on the NPM registry as of May 2021. We were able to mine the name, descriptions, keywords, README, last modification date and repository URL for 1,298,170 packages. As NPM has a 64KB limit on README length on the registry, we also downloaded the full README from corresponding GitHub project, when linked. For 308,887 packages (19.2% of all packages), there was no README available on either site. Next, we excluded empty README files, leaving us with 1,297,678 non-empty READMEs. Finally, we limited our dataset to the 620,221 packages with at least one code snippet, following the process in Section 3.3.2.

As the NPM registry does not contain repository data relating to package popularity, we mined GitHub for additional package data, such as the number of stars and if there was a licence. We then processed the README to count number of lines, markdown code blocks of any language and if there were install or run examples. We determined if there was a run or install example by looking for the headings "install" or "usage", or the use of the commands `npm run` or `npm install`.

3.3.2 Code Snippet Extraction

Code snippets for the code search were extracted from README files. We limited our focus to README files over other documentation as this was the

easiest to systematically mine. It is worth mentioning that NPM recommends all packages include a README (and consequently 80.8% of packages on the registry have a README). Other forms of software documentation do exist, but differ across projects, making it harder to mine. We also chose not to mine repository code for snippets as downloading the source of every NPM package would be very time intensive.

To extract code snippets, we looked for code blocks in the README Markdown. These code blocks often include language information for syntax highlighting, which we used to discard code snippets marked as a non-JavaScript language (e.g., bash commands used to demonstrate package installation or JSON for results of execution). However, as not all code snippets use syntax highlighting, we also analysed the remaining dataset to further filter non-JavaScript snippets. We excluded snippets starting with common commands (`npm install` for example), and used a regular expression to look for JSON objects.

As code snippet extraction is an automated process based on heuristics, we manually verified the results using a statistically representative random sample of 384 READMEs (confidence level of 95% and a confidence interval of 5). Two authors manually compared each README and the resulting code snippets from our extraction process, checking for any missing or non-JavaScript snippets. The authors had a Cohen's Kappa agreement of 0.72 (substantial agreement). Then, the authors met to discuss results and determine final categorisations. Using this data, we refined the extraction process to handle any erroneous cases and reviewed again to verify. Ultimately, we were able to extract 2,161,911 code snippets from the 620,221 READMEs.

3.4 Survey

As a preliminary step in the design NCQ, we conducted a survey to understand how developers find NPM packages and what challenges they experience. The following sections describe our survey design and the results.

3.4.1 Survey Design

To find Node.js programmers, we recruited participants from Prolific[1] –a platform to recruit research participants– and employed a screening survey using a series of programming knowledge questions adapted from Danilova *et al.* (Danilova et al., 2021). Questions were designed to have a single 'correct' answer, so that they could easily be used to verify if the participant had basic programming knowledge. The full details of our screening, including questions and answers, are available in a workshop paper (Reid et al., 2022).

Table 3.2 shows the set of questions we asked participants. Questions were selected to give insights into the frequency of NPM package use, websites used

[1] https://www.prolific.co/

TABLE 3.2. Survey Questions (Q)

Q1	How often do you Search for NPM packages?
Q2	What site do you use the most when searching for NPM packages?
Q3	What kind of challenges do you have searching for NPM packages?
Q4	What code editor do you use the most to program in Node.js?

to find packages and the challenges developers experience. In addition, to understand the way developers program in Node.js in general, we asked about code editor usage (Q4).

We evaluated the *credibility* of participants using the framework proposed by Rainer and Wohlin (Rainer and Wohlin, 2021). We aimed to recruit 'performers', in this case, software developers with real, relevant and recent experience according to the R^3 model. We used the programming skill questions to verify *real* and *relevant* Node.js programming skill, and participants were asked how often they programmed in Node.js to assess that the experience was *recent*. Furthermore, we used Q1 to again assess relevancy of experience in regard to package search; participants that answered 'Never' were not shown Q2 and Q3.

We received 680 responses, out of which 55 answered all questions correctly and were determined to be Node.js programmers.

3.4.2 Survey Results

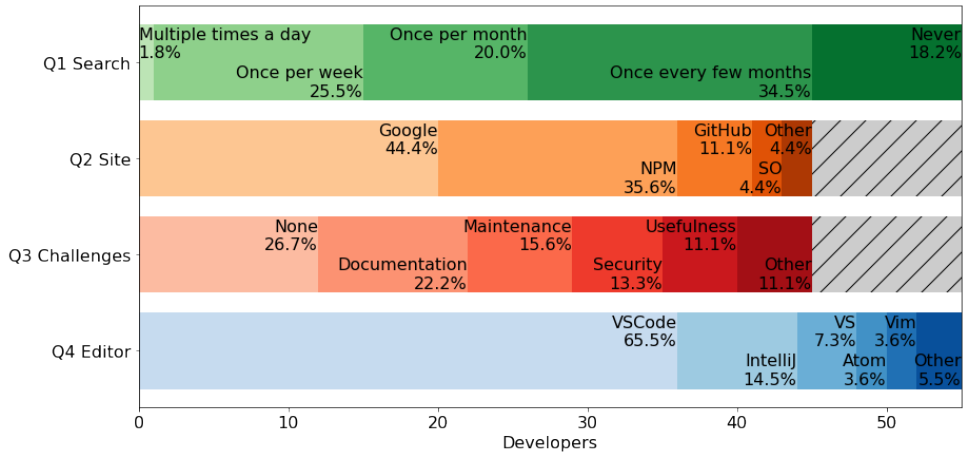

FIGURE 3.7. Responses to survey questions for 55 developers. Q2 and Q3 record no response for developers who do not search for NPM packages.

Figure 3.7 shows the results of our survey. We found that out of 55 developers, 45 searched for NPM packages (Q1) once every few months or more often (81.8%). Considering these developers, the most popular websites to search for packages were Google (44.4%) and the NPM website (35.6%), with most developers (73.3%) reporting that they experienced challenges. The most common challenges mentioned were insufficient documentation and example code

(22.2%), followed by maintenance (15.6%) and security issues (13.3%). Developers highlighted issues finding packages outside the most popular, with one developer saying "Maybe the biggest grip[e] is trying to find packages outside of what's popular that are well documented". In general, most developers had issues finding high quality packages with so many options on the NPM registry; as stated by one developer "I also have a hard time finding out what all the alternatives are to see what my options are". The results for Q4 show that VSCode was the most popular editor for Node.js within our sample. 65.5% of developers answered VSCode, followed by IntelliJ at 14.5%. Therefore, we use it as a baseline for the study described in Section 3.10.

In summary, package use was a common part of programming in Node.js, but most developers had challenges finding high quality NPM packages.

3.5 Overview

Tool	Baseline
$\text{NCQ}_{Retrieval}$	NCQ_{Basic} + Web
NCQ	VSCode + Web

TABLE 3.3. Comparison of baselines used in each evaluation.

To address the problem described in Section 3.2.1, we developed Node Code Query (NCQ). NCQ is a command line tool for Node.js that integrates code and package search into a REPL development environment. The tool extends the existing Node.js REPL and, along with the ability to execute code, adds the ability to search for NPM packages using natural language queries, find and run example code snippets and install and uninstall packages, all without leaving the environment. We first developed an initial version of NCQ, $\text{NCQ}_{Retrieval}$, which focuses on retrieving packages and code snippets, then, using feedback from a user study of 10 developers, we refined the tool further, adding additional features, described in Section 3.9.

Table 3.3 shows each version of the tool and the baseline used to evaluate against. For the first user study, we evaluate $\text{NCQ}_{Retrieval}$ against a baseline version of the tool with no search features and web access; we justify this decision as evaluating the impact of $\text{NCQ}_{Retrieval}$'s search features. For the second evaluation, we then compare NCQ against a more realistic baseline of VSCode and web access, to see how the tool performs against 'typical' processes. NCQ contains all the features of $\text{NCQ}_{Retrieval}$, with additional enhancements such as code correction to reduce time spent fixing errors and package search based on runnability and not popularity. We do not evaluate the two variants against each other as this research focuses on the comparison between automated and manual code reuse techniques.

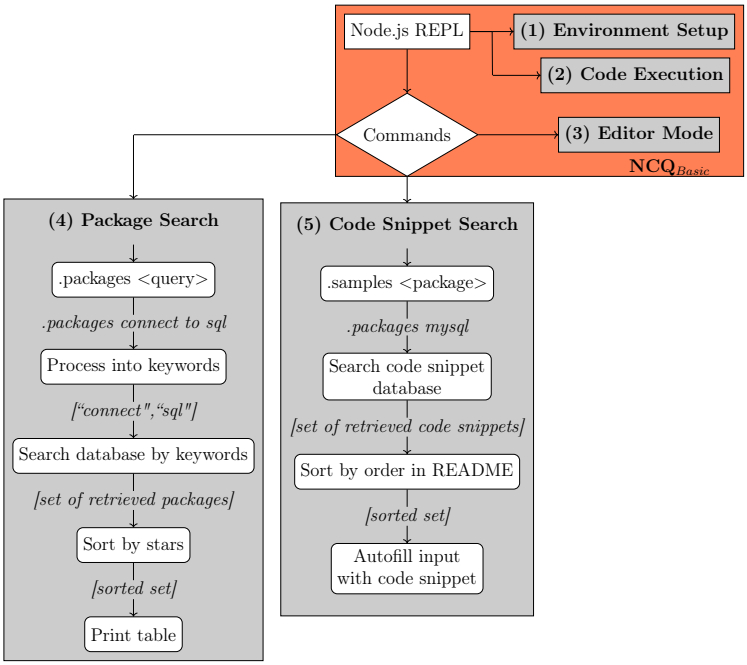

FIGURE 3.8. Overview of $NCQ_{Retrieval}$'s features. Features shared with NCQ_{Basic} are highlighted in orange.

3.6 $NCQ_{Retrieval}$

The initial version of NCQ, $NCQ_{Retrieval}$, focuses on the problem of finding packages and code snippets. As finding and testing packages is often a process with many small tasks where developers may want to quickly try out different packages, we implement $NCQ_{Retrieval}$ as a Read-Eval-Print-Loop (REPL). The REPL model is highly suited to this type of programming; developers receive immediate feedback on execution of each line, and writing and executing code are tied together. Figure 3.8 gives an overview of the features in $NCQ_{Retrieval}$; 1) the ability to set up environments, 2) code execution, 3) the editor mode, 4) package search and 5) code snippet search.

3.6.1 REPL

Starting the REPL creates a new environment to execute Node.js code and install packages. The tool creates a new directory and the necessary project files that allow a user to install packages: (1) the `package.json` file, which stores project details and directly installed packages, and (2) the `package-lock.json`, which lists the entire dependency tree. These files are automatically updated by the package manager when installing/uninstalling packages.

3.6.2 Editor Mode

To make programming in a REPL easier, $NCQ_{Retrieval}$ includes an additional editor mode that allows users to edit the state of the REPL in a traditional editor. The default Node.js REPL maintains a context of executed code as

users submit code, but this cannot be modified besides overwriting previously defined functions and variables. This often means that users need to rewrite entire functions to fix errors. $NCQ_{Retrieval}$'s editor mode provides a simple text editor that presents a user with their previously run code, where they can edit or remove lines. On saving the file, the REPL state will be reset and the new code will be run.

3.6.3 Package Search

NCQ's package search relies on a database of NPM package information, as described in Section 3.3. Packages are pre-processed to create a set of keywords they are indexed by. First, package descriptions are separated into unique words, then combined with the package's keyword field, processed to remove stop words (common, individually meaningless words like "the" and "to"). Then, the remaining set of words are stemmed using the Porter Stemmer algorithm.[2]

```
.packages connect to sql
```

index	name	desciption	stars
0	hapi-postgres-connection	A connection (pool) to PostgreSQL available anywhere in your hapi application	30
1	connect-mssql	SQL Server session store for Connect/Express	27
2	@cloudflare/db-connect	Connect your SQL database to Cloudflare Workers.	22
3	dbq	terse node-mysql query wrapper to ease parallel & series execution, + simple CRUD ops	9
4	sequelstore-connect	An express-session store for Sequelize.js	3
5	fastify-mariadb	Fastify MariaDB connection Pool plugin	3

FIGURE 3.9. Output of the `.packages` command.

A similar process is taken for queries using the `.packages` command, which Figure 3.8 illustrates. For example, the query "connecting to sql" is first separated into the words "connecting", "to", and "sql", then stop words are removed, leaving only "connecting" and "sql". After stemming, the set of words is "connect" and "sql". For each of these words, a set of associated packages can be retrieved; the final result is the subset of packages present in all sets, which is sorted by stars then tabulated with information such as package stars and descriptions to help users make a choice (Figure 3.9).

[2] https://tartarus.org/martin/PorterStemmer/

3.6.4 Code Snippet Search

$NCQ_{Retrieval}$ includes a code snippet search that allows developers to look for example code snippets for a package. Each snippet in our dataset has an associated package it originates from and an index based on the order of appearance in the original README.

As shown in Figure 3.8, using the `.samples` command, code snippets belonging to the given package are returned in order of their occurrence in the original README. Snippets autofill the command line and can be modified or run without needing to copy and paste. We preserve the original order of code snippets to enable developers to more easily understand the logical order some snippets are intended to be executed in; not all code snippets run on their own and they often leave out steps shown in previous snippets, for example, importing the package. Related code snippets also remain grouped together due to this order.

3.7 $NCQ_{Retrieval}$ Evaluation

To evaluate $NCQ_{Retrieval}$ we conducted a user study with 10 Node.js developers, where participants completed programming tasks using the tool. We pose the following related research questions:

RQ1. What is the influence of $NCQ_{Retrieval}$'s search features on the performance of participants?
RQ2. What are the perceptions of participants about $NCQ_{Retrieval}$?

The purpose of **RQ1** is to understand if the tool's search features are helpful, using the comparison baseline of a version of NCQ that lacks search features. Instead, participants used the internet to find packages and examples. We used the following metrics to measure the impact of $NCQ_{Retrieval}$'s search: 1) time to completion, 2) how quickly developers found and installed packages and 3) the number of packages developers tried. The purpose of **RQ2** is to understand developer perceptions of the tool. We analysed participant responses to a set of questions we asked about their experiences after each task.

3.7.1 Experimental Design

This section describes the design of the user study, including how we selected the tasks, what baseline we used, how we recruited participants, how we assigned tasks to participants, how the sessions were run and what questions were asked to participants.

Programming Tasks

As the user study focuses on solving tasks, task selection was important. To represent real programming tasks, we looked at a list[3] of mined Stack Overflow questions from previous work on the NLP2TestableCode project (Reid, Treude,

[3] https://bit.ly/2HK5tLq

and Wagner, 2020). We looked for examples that could be solved in 20 minutes or less by a developer with basic experience in Node.js. It is important to note that, to facilitate the checking of the solutions provided by participants, we modified the original tasks to specify example inputs. For example, to complete task 4, we provide as input to the participant the SQL database to be used so that we could validate the output.

We used the following five tasks:

1. Read a text file (any) from the file system to memory, using a package other than 'fs'. (Sources: Line 9893 of the task list and Stack Overflow question 20186458.)

2. Create an array with ten randomly-chosen numbers and sort the array with "merge sort". (Sources: Line 585 and 99292 and SO questions 42033229 and 2571049.)

3. Use a package to create the CSV file `people.csv` with the contents:

   ```
   Name, Institution, Job
   Alice, Foo, IT Manager
   Bob, Bar, Developer
   ```

   (Sources: Line 3073 and SO question 59600762.)

4. Connect to a MySQL database that we provide and retrieve customers using the SQL command `"SELECT customerName FROM customers WHERE country = 'Australia';"`. (Sources: Line 221 and SO question 6597493.)

5. Create a directed graph in memory with the following edges {(a,b), (a,c), (b,c), (c,d)}. The vertex "a" is the root of the graph. Print the vertices on the console as they are visited in depth-first order. Print each vertex only once. (Sources: Line 13232 and SO question 14483473.)

Participants were said to have successfully completed a task if they produced a JavaScript file that, when run, satisfied the task description. Tasks were designed so that it would be straightforward to determine completion.

Baseline

We designed the baseline of this evaluation as NCQ_{Basic}, a version of $\text{NCQ}_{Retrieval}$ with package and code search suppressed. NCQ_{Basic}'s features are highlighted in orange in Figure 3.8. With NCQ_{Basic}, participants must search for packages online, using the browser. We put no limits on what sites could be used for search, instead of restricting them to just the NPM website, as the results of our survey indicate that developers prefer to use generic search engines like Google to find packages. Participants first solve a task using the baseline, then using $\text{NCQ}_{Retrieval}$, in order to compare the effect of search on results while reducing the impact of tool familiarity. We expect that developers would be more familiar programming in a traditional editor and that this could have an impact on results.

Participants

We hired 10 developers from online, for example, using social media sites such as Twitter and Telegram. We offered participants $40USD and required fluency in English and basic knowledge in Node.js. We estimated the total time of a user session as follows: 10 minutes for presentation/training, 20 minutes to run a task with NCQ_{Basic}, 20 minutes to run a task with $NCQ_{Retrieval}$ and 10 minutes to answer questions after tasks (1hr total). Participants had an average of 2 years experience with Node.js.

Assignment

TABLE 3.4. Fragment of task assignments.

participant	task				
	a	b	c	d	e
1	N	Y	-	-	-
2	-	N	Y	-	-
3	-	-	N	Y	-
4	-	-	-	N	Y
5	Y	-	-	-	N

We used a typical combinatorial design to uniformly assign tasks to participants. For space reasons, Table 3.4 shows a fragment of the design including only 5 participants (rows) and 5 tasks (columns). A cell with the label "Y" (respectively, "N") indicates that a participant will be assigned to solve a given task with $NCQ_{Retrieval}$ (respectively, NCQ_{Basic} + Web). Note that every participant executes one task with $NCQ_{Retrieval}$ and one task with the baseline and, in this design with 5 participants, every task is executed by two participants. The design of this table is a Latin Square as the number of participants and tasks are the same, but the actual design we used included 10 participants to obtain a larger sample. Intuitively, that design can be obtained by repeating the rows associated to participants 1-5 two times. With 10 participants, each task is executed by 4 different participants: 2 participants using $NCQ_{Retrieval}$ and 2 participants using NCQ_{Basic}.

User Session

The user study was undertaken remotely using AnyDesk[4], with participants connecting at an agreed upon time with the instructor (first author of this paper). Before the session, participants were asked to watch a video demonstration of the tool[5]. At the beginning of the session, the instructor demonstrated the features of each tool and then the participant attempted each task. After the participant turned in each task, the instructor asked the participant to complete a section of the questionnaire (see Section 3.7.1).

[4] https://anydesk.com/en
[5] https://youtu.be/C1PZ2g96eVo

Tool documentation (*e.g.*, keys and commands) was made available to participants on the right side of the screen, with the tool itself on the left. Experiments were run within a virtual machine with the necessary infrastructure to run each task provided; for example, for task 4, the database was already set up and a file was provided with the connection details. Snapshots were used to restore the environment to the same initial state for each participant.

Audio and video (screen capture) were recorded for each session. The tool was configured to record timestamped input (use of commands and submitted code) into log files. At the end of each task the user's REPL code was saved into a file. An additional log of visited websites was transcribed from the session recordings.

Questions

Table 3.5. Questions part A, B and C.

A1	How many years of experience do you have with Node.js?
B1/C1	Overall, did you consider the tool helpful to accomplish the assigned task?
B2/C2	Grade your confidence, in a 1-7 scale, that your solution is correct according to the problem specification?
C3	Did you consider the command "packages" useful?
C4	Did you consider the command "samples" useful?
C5	What features did you like about the tool?
C6	What you did not like about the tool?

Participants were asked to complete sections of a questionnaire via a Google Form at the start of their session (Questions A), after the first task (Questions B) and after the second task (Questions C). Table 3.5 shows the set of questions labelled by stage; for questions B1-2 and C1-4 participants were asked to rank agreement on a seven-level Likert scale.

Table 3.6. Features provided to users as answers (F) for C5.

F1	Ability to execute code through REPL
F2	Ability to search for packages
F3	Ability to search for code snippets
F4	Ability of using an editor to modify snippets (without leaving the REPL)
F5	The combination of code search and execution (virtualization) in a single tool
F6	Ability to install and uninstall packages in an environment
F7	Ability to save work and resume from outside the tool (.save)
F8	Ability to access main functionalities through function keys

For each task we asked participants about their confidence in their solutions (B1 & C1) and the helpfulness of the tool they used (B2 & C2). Additionally, in Section C participants were asked about $\text{NCQ}_{Retrieval}$'s features, for example, if

they found certain commands useful (C3 & C4), what features they liked (C5) and what they didn't like (C6). For Question C5, participants were given the list of features seen in Table 3.6 and allowed to select multiple options. The questionnaire concluded with an option to provide suggestions.

Table 3.7. TAM (Technology Acceptance Model) questions.

D1	Using the tool would improve my performance in my job.
D2	Using the tool in my job would increase my productivity.
D3	Using the tool would enhance my effectiveness in my job.
D4	I would find the tool to be useful in my job.
D5	My interaction with the tool is clear and understandable.
D6	Interacting with the tool does not require a lot of my mental effort.
D7	I find the tool to be easy to use.
D8	I find it easy to get the tool to do what I want it to do.
D9	The quality of the output I get from the tool is high.
D10	I have no problem with the quality of the tool's output.
D11	I would have no difficulty telling others about the results of using the tool.
D12	I believe I could communicate to others the consequences of using the tool.
D13	The results of using the system are apparent to me.
D14	I would have difficulty explaining why using the system may or may not be beneficial.

At the end of the session participants answered questions based on the Technology Acceptance Model (TAM) (Davis, 1989), a model for measuring, predicting, and explaining use of technology, including software. Our questions were adapted from TAM2 (Venkatesh and Davis, 2000), which also incorporates job relevance and output quality. These questions can be seen in Table 3.7.

3.7.2 Answering RQ1: What is the influence of $NCQ_{Retrieval}$'s search features on the performance of participants?

Prior research has found that context switching (e.g., switching from a programming environment to a web browser) can disrupt productivity (Proksch, Bauer, and Murphy, 2015). In Section 3.7.1 we describe a version of $NCQ_{Retrieval}$ with no search features; instead, participants search resources online. We compare this to $NCQ_{Retrieval}$, with the hypothesis that the integrated search features will impact participant performance. The following sections elaborate on the research questions (associated with RQ1) that we posed.

Was there impact on time to complete each task?

Figure 3.11 shows duration to solve tasks (in seconds), by participant and by task, using NCQ_{Basic} (left) and $NCQ_{Retrieval}$ (right). All participants solved their tasks. We compare the difference in durations for each participant, and

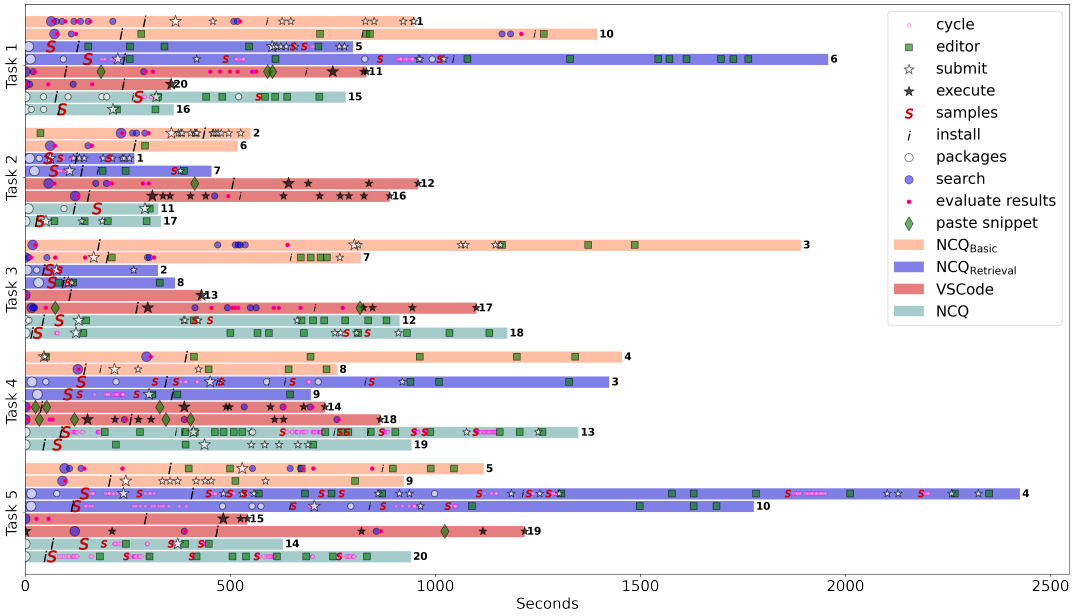

FIGURE 3.10. Timeline of all participants across both user studies, grouped by task and treatment. The x-axis shows time in seconds, markers show features used over time, with the first usage enlarged. Each bar is labelled by participant ID.

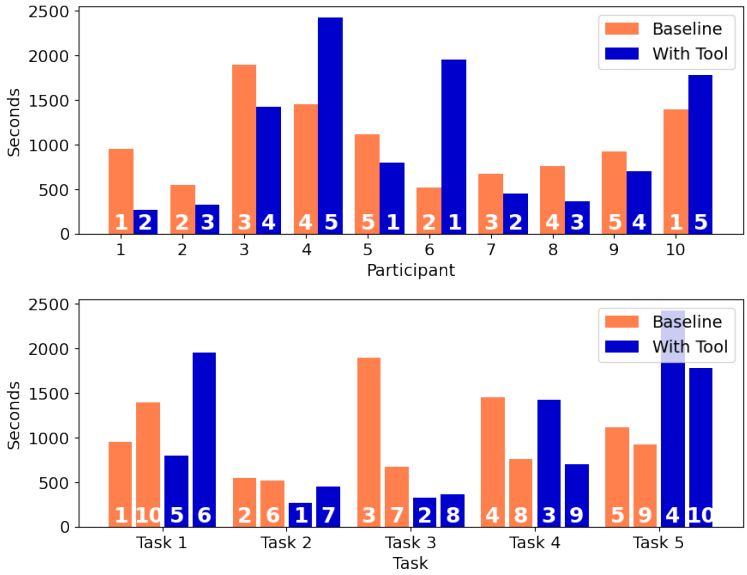

FIGURE 3.11. Duration of tasks by participant and by task. Bars are labelled by task ID in the first figure, and by participant ID in the second figure.

find that 7 out of 10 participants completed tasks faster when using NCQ$_{Retrieval}$ instead of NCQ$_{Basic}$. Participants 4, 6, and 10 were exceptions. Note that each participant solved different tasks. The median time to solve for the baseline was 936 seconds, while the median time for NCQ$_{Retrieval}$ was 748 seconds.

Figure 3.10 shows a timeline of each participant's session, grouped by task and treatment. The figure also shows what features of each tool each participant

used to solve their tasks over time. Looking at the timeline for the three cases of lower $NCQ_{Retrieval}$ performance shows that these participants tried multiple packages, searched for examples multiple times and had large amounts of time between executions. In addition, two of these participants (4 and 10) were assigned the same task (task 5).

Participants solving Task 2 and 3 in $NCQ_{Retrieval}$ were faster than their baseline counterparts, which can be seen in Figure 3.11. Combined with participants 5 (Task 1) and 9 (Task 4), six participants solved their tasks faster using the tool than all participants using NCQ_{Basic} for that task. The reverse was only true for two participants, 5 and 9, who completed task 5 faster using the baseline than both participants using the tool.

To explain the performance difference for Task 5, we can look at the task and how participants tackled it. Task 5 has two associated parts, the creation of a graph and the traversal of nodes in the graph. By looking at the commands participants used to search and their transcribed web searches, we can see that three of the four participants who completed this task focused on the graph creation aspect first, neglecting the traversal of the graph. Only participant 9 using NCQ_{Basic} immediately searched for graph algorithms, focusing on the depth-first search aspect of the task. Naturally, traversing a graph requires the creation of a graph, leading this participant to find a correct package faster than others. Other participants needed to install and experiment with other packages to come to the conclusion that those packages would not help solve the task as not all packages contain depth-first search functionality. Note that the same participant also performed well when using $NCQ_{Retrieval}$ in task 4, where they only needed a single package install per task.

In summary, participants completed tasks in $NCQ_{Retrieval}$ in most cases faster than they did in NCQ_{Basic}.

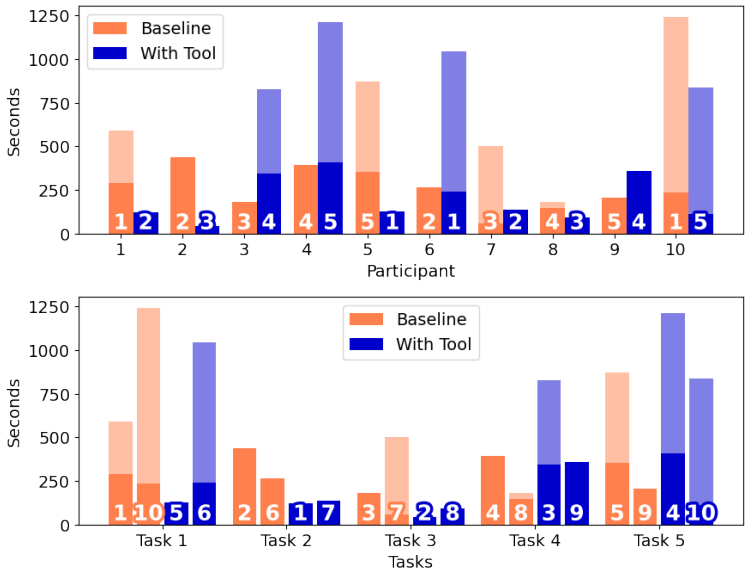

FIGURE 3.12. Time taken to find the first package (darker), and final package (lighter), by participant and by task

.

How long did participants take to install the first and last package?

The purpose of comparing the first and last install is to evaluate how long participants spent evaluating packages. The first use of the `.install` commands marks when participants stop search to try a package, and the last install tells us how long it takes to find a suitable package to complete the task. Where these are the same, it indicates that participants found a suitable package on their first attempt.

As described in Section 3.7.1, we record the use of the `.install` command along with a timestamp. Figure 3.12 shows the time to find first and last package, by participant and by task. Bars of a single colour represent when the first package was also the last package; this is the case for 60% of participants using $\text{NCQ}_{Retrieval}$ in contrast to 50% of participants in NCQ_{Basic}. We can also see that in 60% of cases, participants installed the first package earlier using $\text{NCQ}_{Retrieval}$ as opposed to NCQ_{Basic}. Again, in 60% of cases participants installed the final package earlier in $\text{NCQ}_{Retrieval}$ than they did using NCQ_{Basic}. The median times to find the final package for NCQ_{Basic} and $\text{NCQ}_{Retrieval}$ were 250 seconds and 133 seconds respectively. To sum up, results suggest that, compared to NCQ_{Basic}, $\text{NCQ}_{Retrieval}$ enables participants to begin programming faster and to conclude tasks faster.

Summary: Participants that used $\text{NCQ}_{Retrieval}$ completed tasks without ever leaving the command line interface of the tool. Furthermore, participants using $\text{NCQ}_{Retrieval}$ completed tasks at least as quickly and correctly as they did in the baseline setting (NCQ_{Basic} + Web) where participants had access to web resources such as Google and Stack Overflow.

3.7.3 Answering RQ2: What are the perceptions of participants about $\text{NCQ}_{Retrieval}$?

The aim of RQ2 is to identify developer perceptions about $\text{NCQ}_{Retrieval}$ and its features, such as perceived usefulness which can be important for determining developers' acceptance of new tools (Davis, 1989). The following sections elaborate on the research questions (associated with RQ2) that we posed.

What was the participant perception of $\text{NCQ}_{Retrieval}$'s features?

For this question we investigate what features of $\text{NCQ}_{Retrieval}$ developers found most useful, and respectively, less useful. To measure this, we asked developers a series of questions about $\text{NCQ}_{Retrieval}$'s features after their session, described in Section 3.7.1.

First, we observed how developers used the features of $\text{NCQ}_{Retrieval}$. Figure 3.13 shows how many times developers used each feature while solving tasks in both the tool and baseline. Recall that the commands `.samples` and `.packages` are available only in $\text{NCQ}_{Retrieval}$. Overall, the most used feature was the editor mode, used to write code, followed by the feature to retrieve

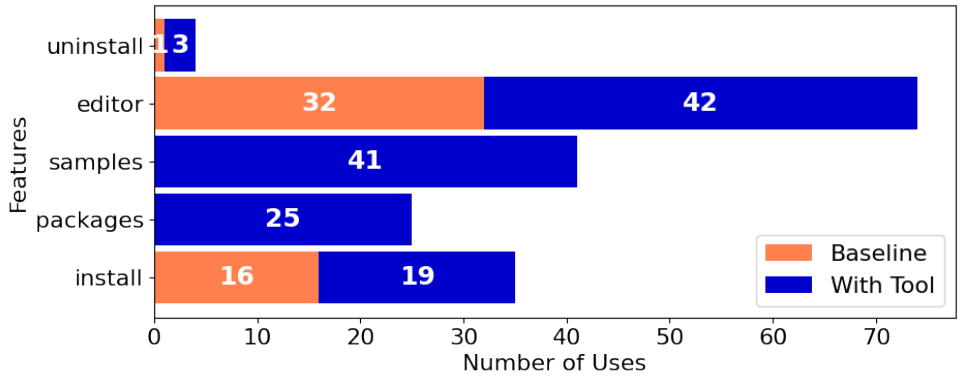

FIGURE 3.13. Usage of $NCQ_{Retrieval}$ and NCQ_{Basic}'s features

samples. In $NCQ_{Retrieval}$, developers looked for packages 25 times, installed a package 19 times and looked for samples 41 times.

For question C5, participants were presented with a list of features shown in Table 3.6, and asked to select what they liked, where multiple selection was allowed. Figure 3.14 shows how many participants selected each feature. All participants found the code snippet search (F3) to be useful, with the package search (F2) and editor mode (F3) also being well liked. These results show that the *search features of* $NCQ_{Retrieval}$ were the highest rated out of all features, and suggest that almost all participants liked the core features of $NCQ_{Retrieval}$.

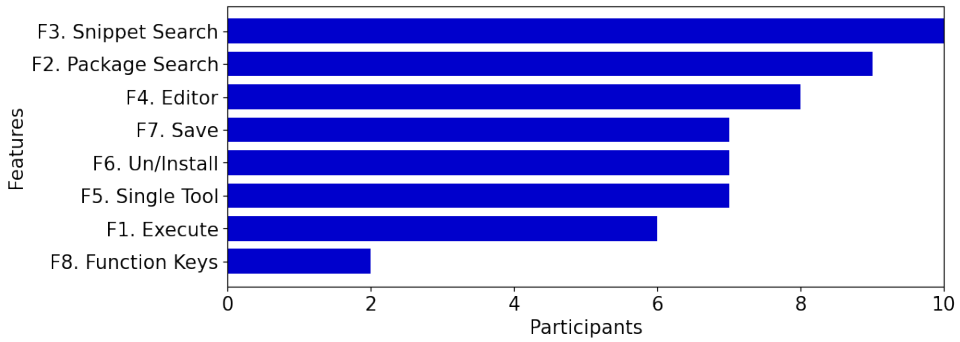

FIGURE 3.14. What features (Table 3.6) participants liked.

Similarly, we asked participants to rate the usefulness of specific commands. Figure 3.15 shows how participants rated the `.packages` and `.samples` commands using a seven-rank Likert scale. Overall, participants found both features to be useful.

We did not ask participants to explain their answers, however, during the session participants indicated issues accessing the function keys from Mac systems, which may explain results for F8 (see Figure 3.14). During the feedback question, the participant who did not like the packages command indicated that the tool should "focus on samples". When asked for feedback at the end of the questionnaire, participants provided suggestions for extending $NCQ_{Retrieval}$'s

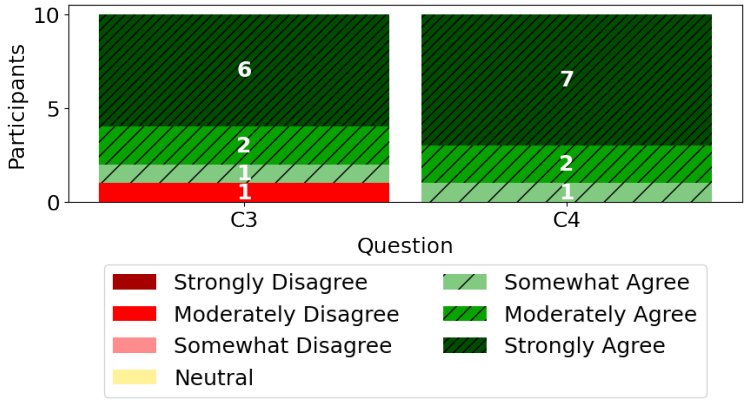

FIGURE 3.15. Participant ranking of the usefulness of the commands "packages" (Question C3) and "samples" (C4).

features, such as better package ranking, more interactive UI, and implementing the tool as a plug-in for an existing editor such as Visual Studio Code.

While the editor mode was highly used and well rated, multiple participants indicated that it was limited, lacking familiar controls, and that they would prefer the tool open an external editor instead, with one participant suggesting "this tool could have a vim mode".

What was the general participant perception of NCQ$_{Retrieval}$?

This question investigates the general perceptions of NCQ. We answered the question in two parts. First, we asked general questions about participant confidence and the helpfulness of NCQ$_{Retrieval}$ and NCQ$_{Basic}$. Then, we used a popular procedure (Davis, 1989; Venkatesh and Davis, 2000) used to evaluate acceptance of technology to guide our questions to participants.

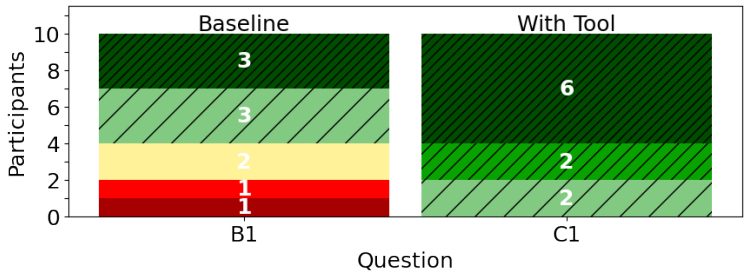

FIGURE 3.16. Helpfulness ranking of tool and baseline.

General questions: We asked participants to rank how helpful each version of the tool was on a seven-rank Likert scale (B1 & C1). We also used the same method to measure how confident participants were in their solutions in each tool (B2 & C2). Figure 3.16 & 3.17 show the responses to these questions. Participants found NCQ$_{Retrieval}$ considerably more helpful than the baseline; all

participants agreed that $NCQ_{Retrieval}$ was helpful, compared to only 60% for the baseline.

Considering confidence of solutions, results were similar between tools, with most participants confident in their solutions, except for 2 participants answering "neutral" for $NCQ_{Retrieval}$. There are many factors that may have influenced these results. We expected developers to be less confident doing something unfamiliar, in this case, searching for packages and examples using $NCQ_{Retrieval}$. Additionally, all participants completed their tasks successfully in both treatments suggesting that we may have overestimated the time budget. There is also the fact that the design of both tools allows participants to execute code before turn in. These impacts may have levelled the confidence observed within the usage of each tool.

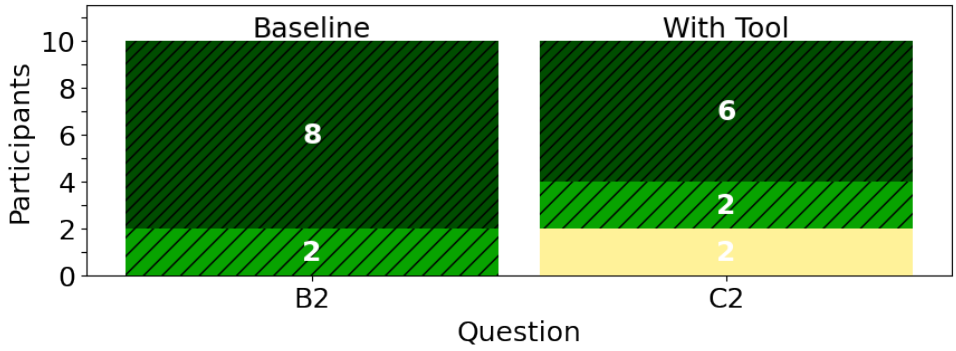

FIGURE 3.17. Confidence in task solutions.

TAM: In addition to assessing developer perception of features, we also surveyed participants using 14 questions adapted from TAM2 (Venkatesh and Davis, 2000) (Technology Acceptance Model), detailed in Section 3.7.1. Figure 3.18 shows the responses for each of these questions.

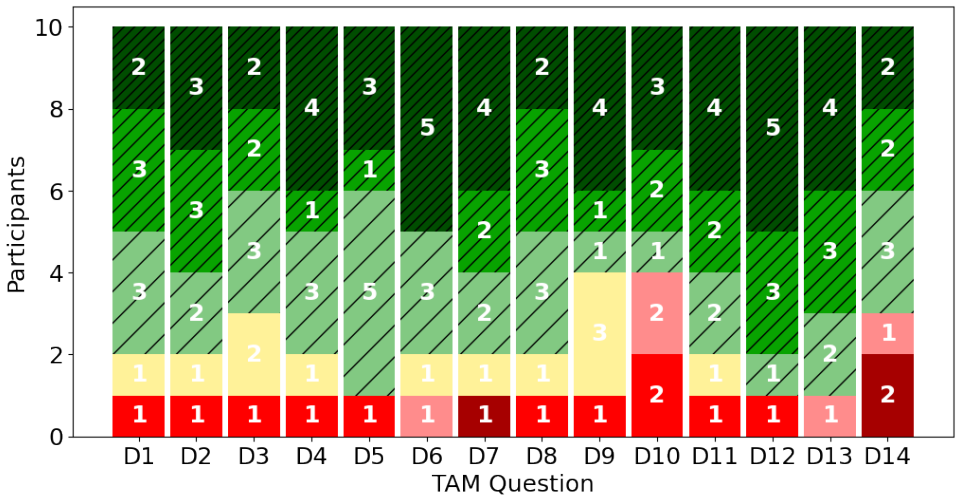

FIGURE 3.18. Participant response to TAM questions.[6]

Overall, developers found NCQ$_{Retrieval}$ to be useful, and their experiences with the tool were positive. For question D14, most participants agreed they would be able to explain how the tool could be beneficial, One participant responded to all TAM questions negatively; this participant also indicated for question C6 that they found the tool "a little difficult to navigate".

Summary: Participants' perceptions about NCQ$_{Retrieval}$ were positive across all evaluated aspects. All 10 participants agreed that NCQ$_{Retrieval}$ was helpful to them in accomplishing the assigned task (compared to 60% for NCQ$_{Basic}$). Searching for code snippets and packages were reported as most popular features, and ratings for perceived usefulness, ease of use, and likely future use (measured by the Technology Acceptance Model) were comprehensively positive.

3.8 NCQ$_{Retrieval}$ Limitations

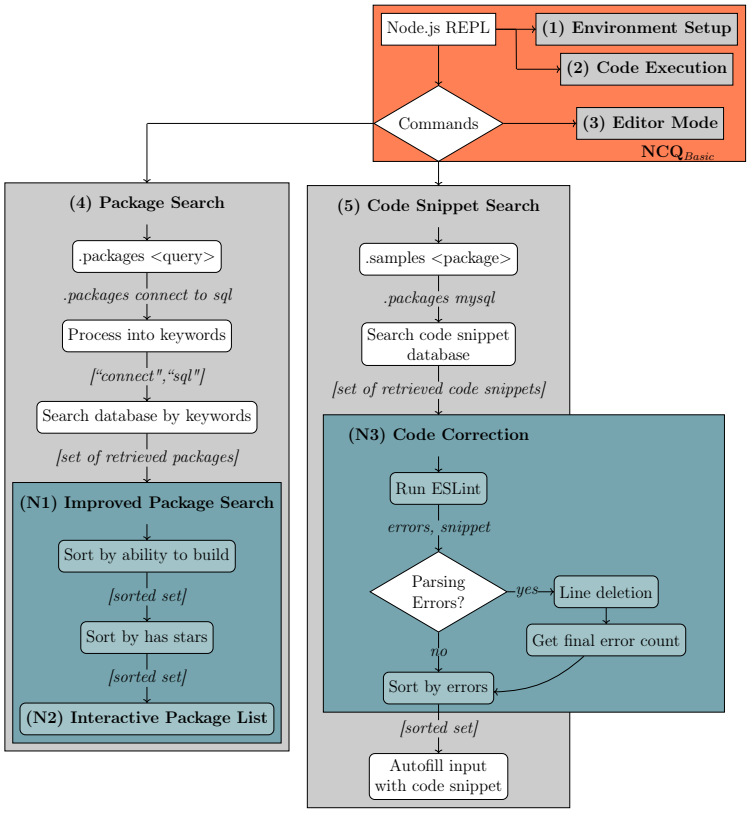

FIGURE 3.19. NCQ's features. The new features of NCQ, in comparison to NCQ$_{Retrieval}$, appear in blue. Note that no NCQ$_{Retrieval}$ functionality was removed for NCQ

This section discusses the limitations of NCQ$_{Retrieval}$, and how a revised version of the tool, NCQ, addresses those limitations.

[6]Scale reversed for D14.

The findings from our user study provides evidence that $\text{NCQ}_{Retrieval}$ enables developers to search for and experiment with different packages with reduced context switching. We observed that developers that use $\text{NCQ}_{Retrieval}$ are at least as quick and successful in their coding tasks as developers that followed a "traditional" setup with access to search engines, the NPM website, and software documentation from any web resource. We also observed that participants' perceptions are positive across all evaluated aspects, with all participants finding $\text{NCQ}_{Retrieval}$ helpful in accomplishing their tasks.

However, the user studies we conducted revealed that $\text{NCQ}_{Retrieval}$ can be improved in important ways. The following list shows how we addressed user feedback.

- Participants expressed that packages highlighted by the existing package search were often not the best choice, as $\text{NCQ}_{Retrieval}$ only uses GitHub stars to rank packages. The results of our user study suggest that when participants cannot find good packages immediately, and instead have to try multiple packages, they spend more time on their tasks. **Action:** We address this in NCQ by improving the package search ranking algorithm by incorporating a measure of runnability, (see Section 3.9.1).

- Multiple participants expressed that the UI could be improved, especially the package search UI. Participants mentioned that the table of 25 packages was visually noisy, with too many packages, and that $\text{NCQ}_{Retrieval}$ could be improved by becoming more interactive. In the user study, we observed that most uses of the `.packages` command were followed by the `.install` immediately after, *i.e.*, participants installed packages immediately after the search. Participants either copy and pasted the package name as part of the `.install` command, or referred back to the table. **Action:** To address these issues, we replaced the table UI with an interactive, scrollable list of packages that takes up less space, and where users can select a package from the list to install (see Section 3.9.1).

- We observed that most code snippets participants used in both techniques did not run (Section 3.10.4) and participants spent a lot of time making code snippets runnable. **Action:** To address this, we implement automated code corrections to reduce the need for developers to make these changes. We also use the error information to rank code snippets (see Section 3.9.2).

- NCQ also adds other, smaller features based on user feedback and real developers' usage patterns, to make the UI more intuitive. We detail these features in Section 3.9.3.

3.9 NCQ

This section presents NCQ, a new version of our tool incorporating some of the features requested by the participants from the previous user study. We focus on two important observations: 1) package search is an important part of

programming in Node.js and 2) much of the code reuse process depends on modifying and bug-fixing code snippets. From those observations, we (1) improve the package ranking in our search to focus on runnability, not just popularity, (2) improve the package search with a more interactive UI, (3) implement measures of code quality and error correction and (4) make common ways developers use the tool more intuitive. Figure 3.19 provides an overview of the additional features of NCQ; blue rectangles highlight the new features. NCQ retains all the features of $\text{NCQ}_{Retrieval}$, accessible in the exact same manner, and the additional features do not add any significant performance overhead.

3.9.1 Improved Package Search

In Section 3.8 we discuss the limitations of $\text{NCQ}_{Retrieval}$'s package search. We improve upon the star-only ranking by implementing a measure of runnability for each package; we used an approach described in previous work (Chinthanet et al., 2021) to predict the ability to build each package (i.e., install dependencies and prepare working environment) as a proxy for its usability. This approach uses a random forest machine learning model and eight package features from our dataset to predict whether the package is able to build: if the package has a licence, if there was a README file, the number of lines and markdown code blocks in that README, the number of Node.js code snippets, if there was a run and install example in the README, and the last update. As Figure 3.19 shows (box "Sort by ability to build"), the random forest's prediction probability of the ability to build is then used to rank packages in search results. It is worth noting that all packages with no stars are moved to the end of results. We found that 53.7% of packages in our dataset had no stars, thus this feature enables us to maintain an aspect of (un)popularity in our search.

To address the limitations of the previous package search UI, NCQ implements an interactive package search. The `.packages` command returns a set of packages; printing a list with all packages on the terminal would make it hard for users to navigate. Our previous solution was to show a subset of packages in a table. NCQ now uses a scrollable list, with terminal height, to show more packages in a smaller space. This interactive design also enables the user to "select" a package for installation upon pressing the enter key.

3.9.2 Code Correction

In Section 3.8 we identified that most code snippets needed fixes to run. To address this, we implement automated code corrections based on similar work in NLP2TestableCode (Reid, Treude, and Wagner, 2020). While our previous work focused on Stack Overflow snippets and Java, code correction for JavaScript is a unique problem.

Unlike Java, JavaScript is not a compiled language, so existing approaches to identify errors like compiler usage cannot be re-implemented here. Instead, existing work in JavaScript error analysis has used the popular linter ESLint (Campos et al., 2019). As a linter, ESLint has code style rules, however, what errors are reported can be customised and it has an automated fix API.

We configure ESLint to report three categories of rules: errors for rules that could indicate unrunnable code (for example, the `no-dupe-args` rule), warnings for other erroneous code that does not stop execution but has fixes (e.g. the rule requiring `===` use, which automatically replaces `==` where necessary) and finally a select number of style rules as warnings to enforce a consistent style to all code snippets in NCQ (indents and semicolons). We use the warning severity to trigger ESLint's fix API without counting minor issues as errors. We find that with this configuration, 45.2% of code snippets in our dataset have linting errors, of which 77.2% are parsing errors which ESLint cannot automatically fix.

In addition to existing ESLint rules, we implement our own. One common error was the use of import/export statements, which the REPL does not support. When running the parser using the `sourceType` "script", to match the REPL functionality, this code triggers a parsing error which cannot be fixed using ESLint. However, if we enable the "module" mode, this no longer causes an error. Instead, we implement a custom ESLint rule for this code. We also implement a custom fix that replaces imports with `require` statements, which most packages also support. With this fix, the error rate reduces to 35.1% of code snippets. Finally, as most errors were parsing errors with no fixes, we implement a line deletion algorithm similar to NLP2TestableCode. We comment out lines where parsing errors occur until no more changes can be made or the code snippet has no errors. The result is that 94% of code snippets have no errors, however 54.2% of snippets fixed by the line deletion algorithm have all lines commented out. To address this, we add an additional sorting mechanism that places these 'comment-only' code snippets at the end of the search results.

Figure 3.19 demonstrates how the error analysis and correction work together. When users search for snippets, each one is run through ESLint's automatic fixes and given an error count. If any of these errors are parsing errors, the line deletion algorithm is run, before a final error count is determined. Finally, the snippets are sorted by error count to prefer error-free snippets.

3.9.3 Additional Features

We add some additional features based on common use cases we saw in the user study. We observed that participants used the editor mode often, and some even pasted snippets directly into the editor. To automate this, opening the editor with empty REPL state now asks the user if they would like to load the previously seen snippet. We also observed that some code snippets with constant variables, often used when importing a package, caused issues for participants when trying to rerun snippets. This would cause cases where code ran once but not again. To help with this, we renamed the default `.clear` command to `.reset` (to differentiate it from "clearing the input") and included an additional error message for this case reminding them they could reset the state of the REPL.

3.10 NCQ Evaluation

To evaluate NCQ, we conducted a user study involving 10 Node.js developers. This time, we compared use of NCQ to the current state of practice, a traditional editor (Visual Studio Code) and online search, instead of an artificial baseline. Section 3.5 describes the baselines for each evaluation. Similar to the previous evaluation, participants were asked to complete two coding tasks with and without NCQ and, after each task, they were asked to answer questions about their experience.

We ask the following research questions about participant performance and perceptions, similar to RQ1 and RQ2, as well as an additional question to analyse NCQ's new code correction feature:

RQ3. What is the influence of NCQ's search features on the performance of participants?

RQ4. What are the participants' perceptions about NCQ?

RQ5. How effective is NCQ's code correction?

3.10.1 Experimental Design

We maintain the same design as the first study, except where necessary changes were made due to the use of a traditional editor. Participants were given the same tasks, with the same assignment and most post-task questions remain unchanged. This section describes the differences:

Baseline

We compare NCQ to the baseline of a traditional editor and internet access. The aim is to measure how NCQ compares to a more realistic baseline consisting of developers programming in a real editor and using online search to find packages and examples. We selected Visual Studio Code, the most popular editor among Node.js Developers, considering the responses we obtained in our survey (Section 3.4). We setup VSCode as a clean installation with no extensions, open to a new Node.js project and empty "index.js", with the integrated terminal open for executing Node.js code.

Participants

Due to difficulties finding participants for the first user study, we decided on a more rigorous recruitment process, using the Prolific platform to recruit 10 participants. We verified that participants had basic Node.js skills using a screening survey where participants were asked programming questions, which we describe more in-depth in a workshop paper (Reid et al., 2022). Participants had an average of 3 years experience. The recruited participants are a subset of the same 55 developers who answered our initial survey questions in Section 3.4.

Questions

Participants were asked to answer questions after each task. This time, we already had their demographic details from recruitment, so omitted section A. We also omitted question B1 (helpfulness of the baseline tool) as participants now use VSCode for their first task. Questions can be seen in Table 3.8.

Table 3.8. Questions asked (B) after task 1, and (C) task 2.

C1	Overall, did you consider the tool helpful to accomplish the assigned task? (1 (Strongly Disagree) - 7 (Strongly Agree))
B1/C2	Grade your confidence, in a 1-7 scale, that your solution is correct according to the problem specification?
C3	Did you consider the command "packages" useful?
C4	Did you consider the command "samples" useful?
C5	What features did you like about the tool?
C6	What you did not like about the tool?

At the end of the session participants were asked to answer the same TAM questions as in the first evaluation.

3.10.2 Answering RQ3: What is the influence of NCQ's search features on the performance of participants?

The goal of RQ3 is to identify the helpfulness of NCQ's features and how those features are used by developers. As in RQ1, we hypothesise that integrating package search within a programming environment reduces context switching, improving developer's performance. To answer this question, we compare how participants performed searching for packages using NCQ and using the internet when programming in VSCode.

Was there an impact on time to complete each task?

Figure 3.20 shows the task durations for both NCQ and the baseline. Again, all participants solved their tasks. Participants completed tasks in NCQ faster than in a traditional editor in 6 out of 10 cases, however, the median time to solve for both were similar: 849 seconds for the baseline and 847 seconds for NCQ.

The timeline in Figure 3.10 details each session, with participants 1-10 belonging to the first user study and participants 11-20 belonging to this study. The icons remain the same as in the first study. Participants using VSCode + Web (note the 'search' marker in the timeline figure) follow links to evaluate the results of their search as opposed to cycling through snippets. In NCQ versions participants can press enter to execute code or use the editor mode, which we label as 'submit' or 'editor' in the timeline. In VSCode, we instead transcribed when participants used the terminal to 'execute' their code, and when they pasted code snippets from online into their open file. However, not

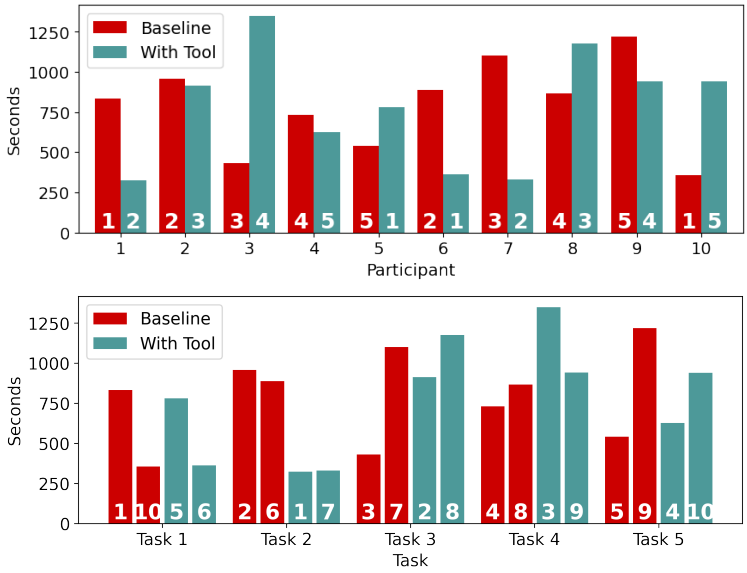

FIGURE 3.20. Duration of tasks by participant and by task. Bars are labelled by task ID in the first figure, and by participant ID in the second figure.

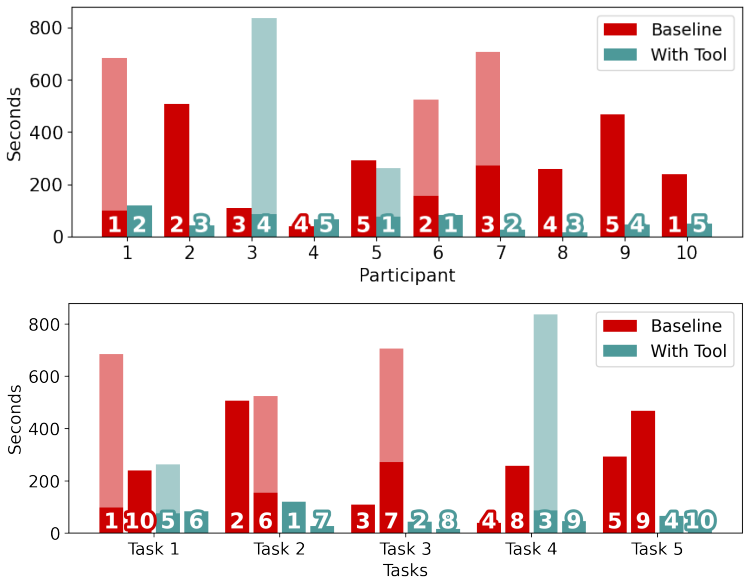

FIGURE 3.21. Time taken to install the first package (darker), and final package (lighter), by participant and by task.

all participants pasted code snippets, instead some manually rewrote code while making changes (e.g. 13, 15, 16 and 20).

The three fastest sessions using NCQ (11, 16, 17) involve only one install and snippet search, conversely, the longest sessions (13, 18, 20) have many uses of the snippet search. Again, as in the previous user study, we observe that the number of times participants search may negatively impact their task duration.

How long did participants take to install the first and last package?

Figure 3.21 shows the times the first and last packages were installed. This information reveals how long participants take in the "package search" phase of their tasks. We observed that the final installed package was the package used to complete the task in all cases. Participants' first and last packages were the same (only 1 package installed) in 8 cases for NCQ and 6 using VSCode. In 7 cases participants found their last package faster using NCQ than VSCode. The median times to find the final package for the baseline and NCQ were 380 seconds and 57 seconds respectively. These results highlight the efficiency of finding packages in NCQ.

Summary: Participants completing tasks using NCQ performed better than when using VSCode. They were able to solve tasks and find packages faster, in most cases finding a suitable package immediately.

3.10.3 Answering RQ4: What are the participants' perceptions about NCQ?

The goal of RQ4 is to identify developer perceptions of the tools, such as helpfulness, and confidence in their solutions. We ask the following questions:

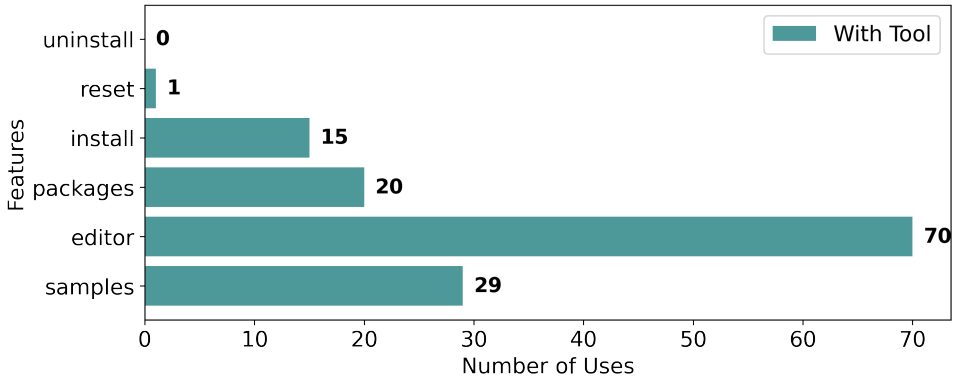

FIGURE 3.22. How many times NCQ's features were used.

What was the participant perception about the features that NCQ offers?

The goal of this question is to understand participant perceptions of each of NCQ's features and to identify which features were useful or not.

Figure 3.22 shows the features that participants used the most. As in the first study, the *editor* mode and the search for *samples* remained the most popular features. Relative to the first study, there was an increase in the total number of uses of the editor mode, but a reduction in the use of the commands `.packages` and `.samples`. In the first case, the reduction may be attributed to the changes to the editor mode whereas, in the second case, the reduction may be justified by the fact that more participants solved tasks faster and with the first package, reducing the need to look at more packages and their examples.

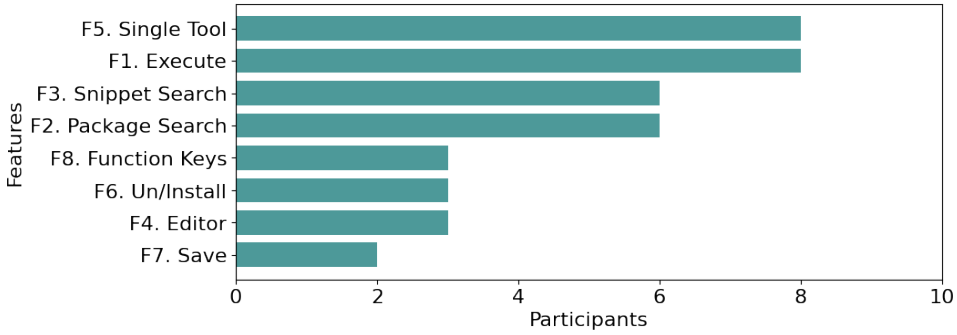

FIGURE 3.23. Number of participants that liked each feature of NCQ. Features (F) are listed in Table 3.6.

Recall that Table 3.6 shows the list of NCQ's features. According to question C5 (see Table 3.8), the features that participants liked the most were the "ability to execute code in the REPL" (F1) and "the combination of code search and execution in a single tool" (F2). The feature participants liked the least was F7, "the ability to save and resume". This is a shift from the first study, where snippet, package and editor mode were the most highly rated. In fact, the editor mode was only selected by three participants despite being used frequently. In general, most participants liked the core features of NCQ: the package (F2) and code snippet search (F3), REPL execution (F1) and combination of search and execution environment (F5).

Figure 3.24 shows how participants ranked the usefulness of commands `.packages` and `.samples` using the same seven rank Likert scale from the previous user study. Eight participants agreed that both features were useful, with only one strongly disagreeing with the samples feature (this participant then answered for question C6 that they thought showing only code snippets and not their surrounding documentation which "exists to guide the user" was limiting). Despite this feedback, all participants were able to successfully complete their tasks in NCQ, and the results of our user study suggests they performed similarly or better than when they had access to this documentation in the baseline. While we cannot conclude that a lack of documentation has a positive impact on performance, in fact, pairing documentation and code snippets may have

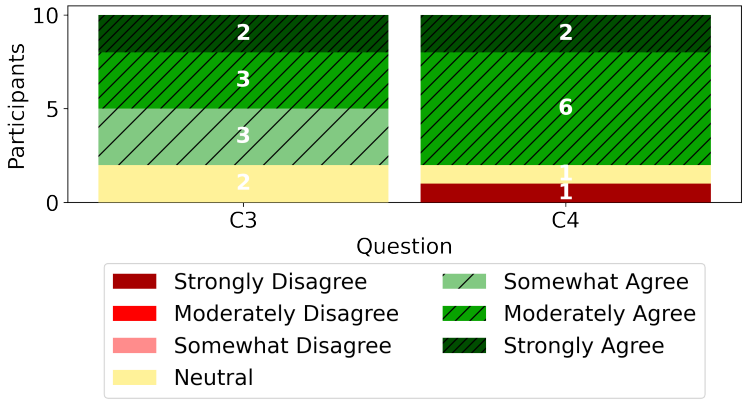

FIGURE 3.24. Participant ranking of the usefulness of the commands "packages" (question C3) and "samples" (C4).

improved performance further, we can argue that participants were not limited greatly because of it. In general, there was a slight decrease in agreement between studies, but the result is still very positive.

What was the general participant perception of NCQ?

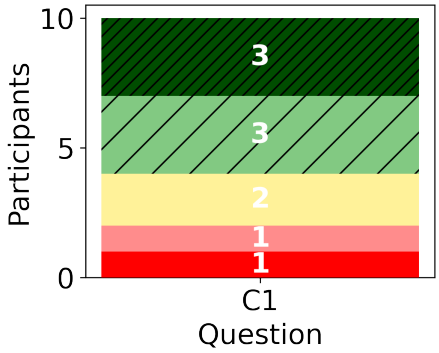

FIGURE 3.25. Participant rankings for helpfulness of NCQ (C1)

We asked participants to rank the helpfulness of NCQ; Figure 3.25 shows the responses. Six out of ten participants agreed that NCQ was helpful in solving their tasks, and only two participants disagreed. Agreement fell compared to the previous study, however. We did not ask participants to rank the helpfulness of the baseline, VSCode, in this task, but we conjecture that the comparison to a more familiar traditional editor may have impacted participant perception of NCQ compared to the previous evaluation.

Figure 3.26 shows the confidence (1-7 scale) of participants in their solutions for each technique. We observe that confidence is similar between techniques; nine participants scored their confidence highly for the baseline (score of 5 or above), compared to eight for NCQ. Individually, four participants had their confidence increase or stay the same, with the largest increase from a score of

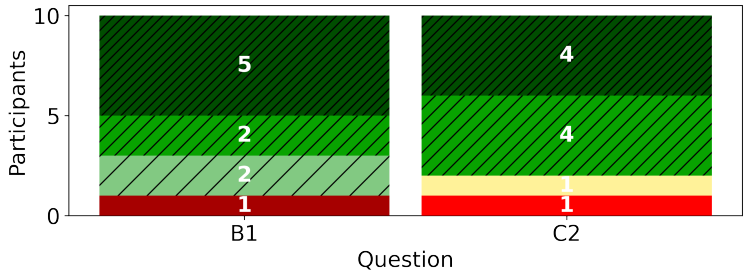

FIGURE 3.26. Participant confidence ratings (B1 & C2).

1 to 7. For four other participants their confidence only fell by one point. This decrease in confidence is also consistent with the first study. Overall, participant confidence in solutions did not suffer considerably despite the use of a new tool when compared to a more familiar baseline.

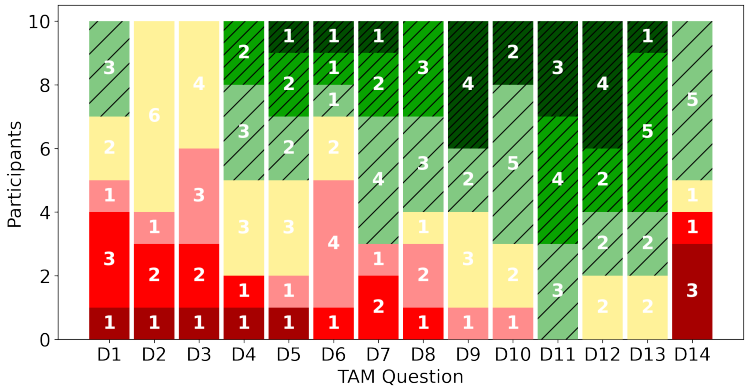

FIGURE 3.27. Participant response to TAM questions.[7]

Participants were asked the same TAM questions as the first study. Results were not as overwhelmingly positive as the first study, however, responses were still more positive than negative, as can be seen in Figure 3.27. All participants agreed with D11 (no difficulty telling others about the results of using the tool), and most participants agreed with D7 (easy to use), D10 (high quality of output), D12 (could communicate to others the consequences of using the tool) and D13 (results of using the system are apparent). However, most participants disagreed with D3 (using the tool would enhance my effectiveness in my job), and six participants agreed with D14, that they would have difficulty explaining why the system may or may not be beneficial.

Summary: Participant perception of NCQ was generally positive; most participants were confident in their solutions and found the tool helpful, despite the comparison to a more familiar baseline, VSCode. Using the Technology Acceptance Model to measure, most participants found NCQ easy to use, with a high quality of output.

[7]Scale reversed for D14.

3.10.4 Answering RQ5: How effective is NCQ's code correction?

The goal of this research question is to evaluate the impact of NCQ's code correction features on code snippet quality. First, we observe that 33% (27/81) of code snippets retrieved by participants using NCQ had errors fixed. To measure the effect of this, we compare the quality of code snippets participants interacted with across all sessions, including the previous user study. We reason that NCQ should improve the quality of code snippets through its code correction features.

For the purpose of this analysis, we define:

- "seen" as the set of all code snippets participants encountered during their task solving, which they evaluate to select snippets to use. For NCQ, we counted all code snippets that participants saw using the `.samples` command, which were automatically logged. However, for online search, we manually logged from video all snippets that were visible in the web browser. Partial snippets (those that the participant did not scroll down far enough on a web page to see entirely) are excluded as we do not have the entire snippet to evaluate. Likewise, we argue that participants could not have evaluated these snippets for use without seeing them in full.

- "attempted use" as the subset of "seen" that participants attempted the code reuse process with; that is, participants copied these snippets into their editor, before making changes and/or running them.

- "successful use" as a subset of "attempted use", where any part of a code snippet was successfully used by a participant to complete their assigned task.

To evaluate the quality of code snippets, we look at both linting and runtime errors. We use the same ESLint configuration used in NCQ to detect linting errors. To measure the presence of runtime errors, for each code snippet, we created a Node.js project, installed any needed packages and ran the code snippet in a file. While existing work has shown that most code snippets in documentation do not execute (Chinthanet et al., 2021), instead many are missing parts and may not be intended to be fully working examples, we still consider this measure to be useful for comparison purposes. Developers reusing example code must still transform unrunnable code into runnable code, and runtime errors may represent work needed. The goal of automated error correction is to reduce this work, making runtime errors a useful measure.

Figure 3.28 compares the two baselines and two NCQ variations from both evaluations. We found that in the first user study, 60% of seen and 55% of used code snippets did not run, and that 22% of seen snippets and 24% of used snippets have linting errors. For the NCQ user study, the runtime error rate remains similar for seen and used at 66% and 64% each, however, we see a large reduction in linting errors with the code correction features of NCQ. While the number of seen snippet linting errors in the baseline remains similar at 12%, code snippets found using NCQ had an error rate of 8%. Even more, none of

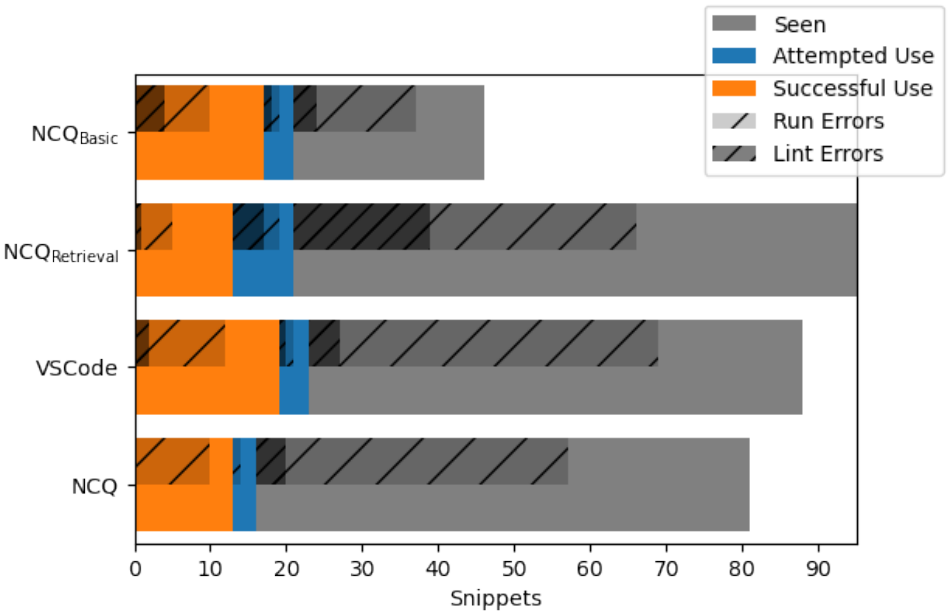

FIGURE 3.28. Code snippet breakdown for each technique.

the snippets participants used had any linting errors. We also see a 5% increase in the proportion of used snippets that were used successfully, which is not seen in the first user study.

Additional observations from this analysis can be made. In the first study, using $\text{NCQ}_{Retrieval}$'s search features, participants were able to see more code snippets more quickly than in the baseline, despite attempted use rate remaining the same. However, in NCQ, we see that participants saw less snippets, and also attempted to use less snippets, than in the baseline, following the trend of participants being more efficient while using NCQ.

For all techniques, we see that there are more code snippets with runtime errors than there are with linting errors. This is a strong indication that there is room for improving the error detection in NCQ, and in improving static analysis in JavaScript in general.

Summary: NCQ's code correction features were employed on 33% of code snippets, and reduced the number of snippets with errors seen and used by developers; all code snippets participants used in NCQ had no linting errors. However, we saw no reduction in runtime errors, indicating there is room for better static analysis of JavaScript code for error detection and thus error correction.

3.11 Discussion

Developers increasingly rely on gluing existing code together as opposed to developing new code (Abdalkareem et al., 2017). Node.js developers in particular have access to a rich ecosystem of 1M+ software packages through NPM. However, navigating this wealth of potential code solutions is a challenge. One user

study participant summed up their experience with: "Sometimes there isn't enough information on each package, and it is very tiring to install/uninstall everything before having a good idea of what it does". Searching and installing packages is a time sink, as described by another participant: "It's not unusual to spend more than a [sic] hour looking for a package for simple job". Yet, installing packages and running corresponding code snippets are crucial activities in deciding whether to use a package. Alleviating these challenges is the goal of NCQ, as summarised by a participant: "Finding a suitable package can be difficult given the volume of packages available. It can become tiresome installing packages and trying out code, constantly referring back to documentation or the dedicated package page on npmjs.com. The `.samples` feature of NCQ looks to reduce the time taken to test packages, to determine if the package provides benefit, and is suitable for a given project." When using NCQ, developers do not need to context switch between editor and search engine; instead they can explore packages, install and uninstall them, and run code snippets from within the same command line interface.

NCQ supports developers' desire to try out code snippets quickly and experiment with different potential solutions when embarking on a new task. Even for trivial tasks, developers commonly use third-party packages (Abdalkareem et al., 2017), and NCQ supports this workflow. We observed that the typical usage pattern participants followed in NCQ was to search for packages, then search for samples within those packages. The example in Section 3.2.2 describes such a workflow based on a real user study session, and we observed similar workflow for online search; most participants started by looking for packages and package documentation on NPM or GitHub, with only a few accessing tutorial sites. This reveals the need for Node.js specific development tools to fit the workflow of Node.js developers.

For NCQ, we improved upon $\text{NCQ}_{Retrieval}$ based on the feedback obtained from users of the tool. The results of NCQ's evaluation indicate that the tool enables developers to be more efficient finding packages than both $\text{NCQ}_{Retrieval}$ and a traditional editor combined with internet access. For example, the average task completion duration reduced from 1,045 seconds for $\text{NCQ}_{Retrieval}$, to 774 for NCQ. However, despite this performance improvement, we see that developer perceptions of NCQ were harsher. There are multiple factors that may explain this. For the first study, most participants were students, and we did not verify Node.js skill; in contrast, for the second study, we made the additional effort to recruit participants with a minimum level of Node.js skill while trying to recruit non-student programmers. This difference in experience between the two groups may impact perceptions of the tools and our ability to compare the two evaluations. Also, the different baselines may have impacted how developers perceived each tool. In the NCQ evaluation we directly contrast the tool to a more familiar baseline; participants use NCQ directly after using VSCode. However, in the $\text{NCQ}_{Retrieval}$ evaluation participants use a baseline that is simply $\text{NCQ}_{Retrieval}$ with reduced features; there may be a perceived 'improvement' between the two tasks, and participants were not faced with a direct comparison between $\text{NCQ}_{Retrieval}$ and tools they use regularly to program. Despite all of this, results did remain similar, and in most cases still

positive.

Our findings provide evidence that NCQ still performs well even when directly compared to a familiar, professionally developed baseline such as VSCode. Our evaluation confirms that NCQ was well received by developers because it reduces the need for context switching typically associated with reusing third-party code.

3.12 Threats to Validity

Similar to other empirical studies, there are threats which may affect the validity of our evaluation of NCQ.

Threats to the *construct validity* correspond to the appropriateness of the evaluation metrics. We evaluated NCQ in terms of participant performance and participant perceptions. Our evaluation approaches, such as the Technology Acceptance Model, have been used before in similar studies (e.g., (Steinmacher et al., 2016)), and our evaluation methodology reflects our goals behind developing NCQ.

Threats to the *internal validity* compromise our confidence in establishing a relationship between the independent and dependent variables. Participants were not informed exactly how versions of NCQ ($\text{NCQ}_{Retrieval}$, NCQ_{Basic} and NCQ) were implemented to ensure that their behaviour and responses would not be biased towards any of the tools. In both evaluations, participants worked on different tasks using both the tool and a baseline, which may have affected their experience and responses to our questions. The demonstration of how to use the tools at the beginning of each session may have influenced how participants used them; we mitigated this by having all participants watch the same demo video before their sessions. Because the study was conducted remotely, we cannot guarantee that participants did not search for solutions outside of the study setup. Participants were asked to use the web to find packages in task 1, however, were free to choose what websites to use; in this case, the choice of search engine, and thus knowledge of what options exist, may have influenced results. The design of our survey and participant screening was intended to minimise the number of non-programmer responses in our data, however, these measures may have had additional impacts on the participant pool.

Threats to *external validity* correspond to the ability to generalise our results. We cannot claim generalisability beyond Node.js or the particular implementations of NCQ and NCQ_{Basic} used in our study. Recruiting more or different developers to participate in the study and asking them to work on different tasks may have led to different results. We cannot conclude that improvements in performance between baseline and tool were due to any specific techniques or the automation of the process itself.

3.13 Conclusion and Future Work

The results of our evaluation indicate that Node.js developers have distinct expectations for a code reuse tool, stemming from Node.js's package-based ecosystem. In general, we found that performance of participants in solving tasks did not reduce when using NCQ compared to a more familiar baseline of VSCode and web access, despite the use of an unfamiliar tool with new commands and controls. We found that the perceptions participants had about NCQ were generally positive, with most participants finding NCQ helpful in accomplishing their assigned tasks. Participants' responses to questions about usefulness, easy of use and likely feature use using the TAM (Technology Acceptance Model) were also mostly positive. We also observed that NCQ's code correction features successfully improved the quality of snippets that participants interacted with.

However, our results also indicate that there is more that can still be done to aid developers in streamlining the code reuse process. Better measures of package quality could be investigated other than popularity and runnability. Future work may also involve implementing NCQ in a different context, for example, many participants mentioned they would like an extension to their existing editor over a new tool. To enable package and code snippet search in an editor, we could implement NCQ as a plug-in for VSCode. Future work could also expand on code snippet sources; the current version of the tool only uses examples from README, however, we found that just 38.6% of NPM packages had code snippets in their READMEs. In these cases, mining API usage examples from NPM package source code may be useful. We observed that when participants tried multiple packages, such as in the examples in Section 3.2, they were looking for similar packages; the package search thus could incorporate a 'similar' package suggestion system, using package data to determine package 'similarity'.

As the Node.js ecosystem is always growing, the ability to update the tool's database would be useful (we use an offline database for efficiency). Currently, to update the database, the mining process must be re-run, but it may be possible to use the NPM registry API to keep the database up-to-date. As existing code search tools include online search, we did not investigate this aspect in this paper.

Acknowledgement

This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, CNPq grant 465614/2014-0, and by ARC grants DP200102364 and DP210102670. Brittany's research was supported by an Australian Government Research Training Program (RTP) Scholarship. The work of Christoph and Markus was supported by gifts from Facebook and Google.

Chapter 4

Correcting Code Examples

Related Publication: This chapter is based on our paper: Brittany Reid, Christoph Treude, and Markus Wagner (2023). "Using the TypeScript compiler to fix erroneous Node.js snippets". To appear in: *23rd IEEE International Working Conference on Source Code Analysis and Manipulation* (Reid, Treude, and Wagner, 2023).

As discussed in Chapter 2, 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, in order to aid developers during the code reuse process: 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.

Code available on Github: `https://github.com/Brittany-Reid/node_code_correction`

4.1 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 (Chinthanet et al., 2021). Because software developers frequently reuse code from online sources (Baltes and Diehl, 2019), 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 (Reid, Treude, and Wagner, 2020; Yang, Hussain, and Lopes, 2016; Reid et al., 2023b; Campos et al., 2019; Licorish and Wagner, 2022a; Licorish and Wagner, 2022b). Code reuse tools such as NLP2TestableCode (Reid, Treude, and Wagner, 2020) and NCQ (Reid et al., 2023b) 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 (Reid, Treude, and Wagner, 2020; Reid et al., 2023b). Additionally, such tools can provide insights on the quality of online code in general: for example, Yang et al. (Yang, Hussain, and Lopes, 2016) looked at the usability of Stack Overflow snippets via static analysis.

Research in Java leverages the compiler for error detection and correction (Terragni, Liu, and Cheung, 2016; Reid, Treude, and Wagner, 2020), 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 (Yang, Hussain, and Lopes, 2016; Reid et al., 2023b; Campos et al., 2019). For example, NCQ (Reid et al., 2023b), a command-line REPL (Read-Eval-Print-Loop) programming environment, which automates the process of reusing code snippets from NPM package documentation, uses ESLint (ESLint, 2023) 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 (Campos et al., 2019). 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 (Microsoft, 2023b) 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 VSCode to

provide error highlighting and fix suggestions for JavaScript as well (Microsoft, 2023c), 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, while TypeScript generates ASTs for all but 230 snippets (0.002%) where compilation either crashed or timed out.
- 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`

4.2 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 4.1 (an unedited Stack Overflow snippet from our dataset) while searching on Google. Like many snippets found online, it has errors.

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 4.5, these are common errors, as example code often omits parts to simplify the snippet.

```
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 4.1. Example code snippet from Stack Overflow answer 45582298.

```
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 4.2. 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 4.2. 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.

```
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 4.3. Excerpt of the manually corrected snippet.

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 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.

4.3 Approach

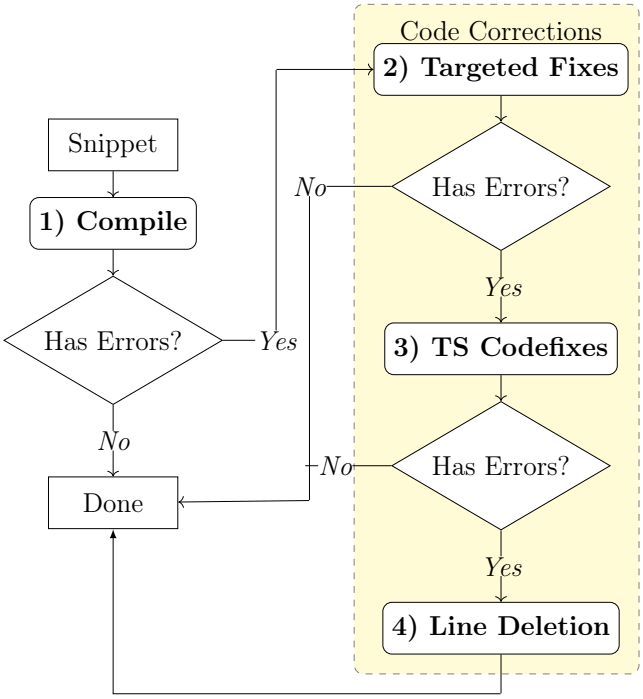

FIGURE 4.4. The NCC pipeline.

Node Code Correction (NCC) has four stages, illustrated in Figure 4.4; 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 built-in 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.

4.3.1 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 (230 snippets), we run the compiler in a separate process with a timeout of 60 seconds.

4.3.2 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 4.2 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.

4.3.3 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 (Microsoft, 2023a), 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.

4.3.4 Line Deletion

Line deletion is a commonly used technique to reduce errors in code snippets (Le Goues, Weimer, and Forrest, 2012; Harrand et al., 2019; Licorish and Wagner, 2022b; Licorish and Wagner, 2022a; Petke et al., 2019; Ginelli et al., 2022; Reid et al., 2023b). 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 (Pham and Yang, 2020). 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,

Algorithm 1: Line Deletion Algorithm

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

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 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 230 crashing or timed out 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.

4.4 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.

4.4.1 NPM Snippets

Reid et al. (2023b) 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 4.2 (a developer looking to reuse code snippets in a regular Node.js programming environment) and report errors before and after fixes.

4.4.2 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 (Baltes et al., 2018; Baltes and Wagner, 2020), retrieving the code-only `PostBlockVersion`s 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.

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

Table 4.1. 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

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 4.1 shows the breakdown of the data.

4.5 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 4.4.1) 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 4.2; 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 4.4.2. We ask the following research question:

RQ5. How does NCC compare to manual fixes?

4.5.1 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.

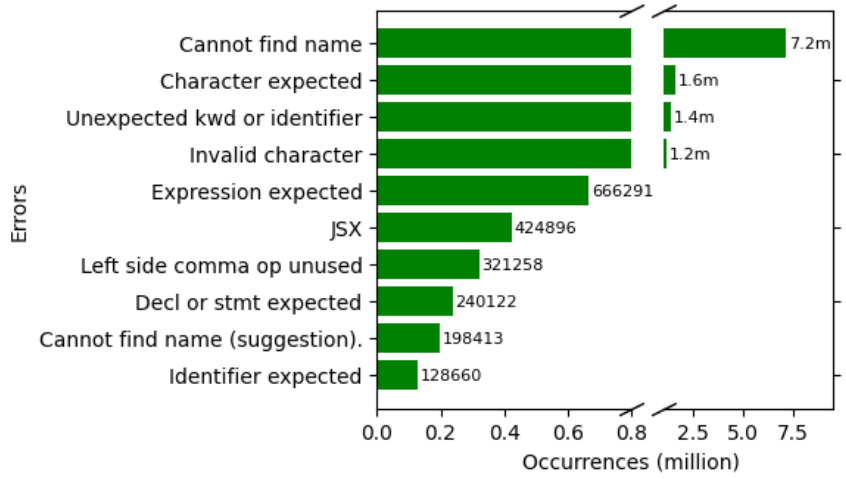

FIGURE 4.5. Most common error types in NPM documentation, reported by TS.

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 4.5. 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 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.

Figure 4.6 shows a common situation in NPM package documentation. These two code snippets from the README for 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

```
var prompt = require('prompt');

prompt.start();
prompt.get(['username', 'email'], function (err, result) {
    console.log('Command-line input received:')
    console.log('  username: ' + result.username)
    console.log('  email: ' + result.email)});
```

```
const {username, email} = await prompt.get(['username', 'email']);
```

FIGURE 4.6. Two code snippets from the package `prompt`.

between code snippets, such as the `require` statement in Figure 4.6. 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.

4.5.2 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.

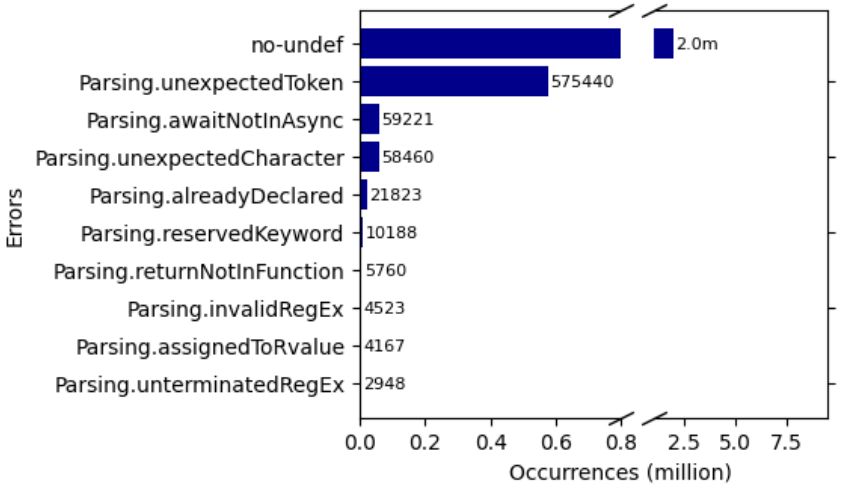

FIGURE 4.7. The 10 most common error types via ESLint.

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.

ESLint reports 185 different error types, of which 175 of the types (94.6%) are parsing errors. Figure 4.7 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 it's intended purpose as a *linter*; further lowing 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.

4.5.3 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 4.8 illustrates the most common error types after codefixes and shows significant changes. All 10 of the most common errors in Figure 4.5 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 4.5.1,

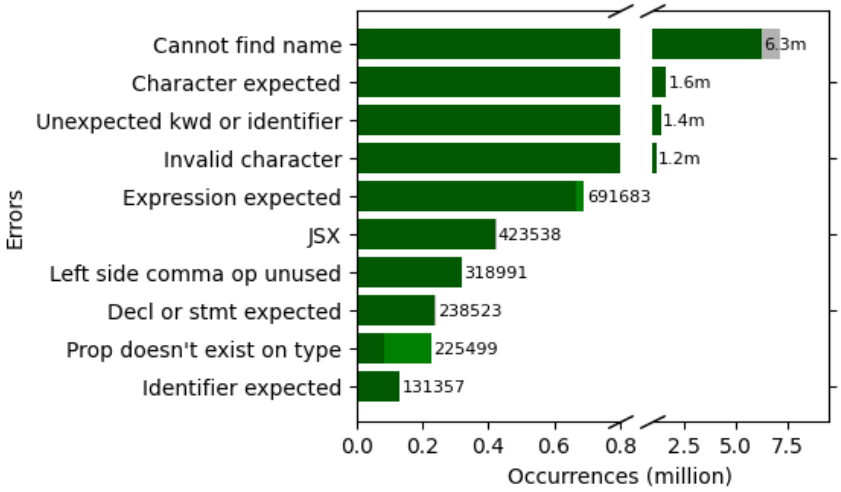

FIGURE 4.8. The 10 most common error types after TS codefixes. Light Grey represents a decrease in errors from previous results in Figure 4.5, where light green represents an increase.

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 commented 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.

The types of errors that were reported changed considerably between codefixes and deletion, as seen in Figure 4.9. 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

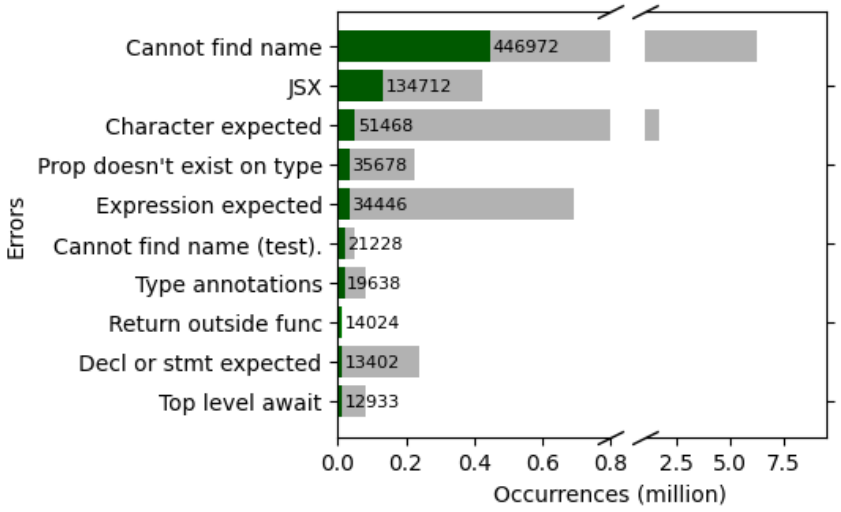

FIGURE 4.9. The 10 most common error types after deletion and TS codefixes. Shading represents improvement over Figure 4.8.

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 4.10. 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 4.10 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 4.11 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 not constructable` also handles a similar case, where the previously undefined identifier should instead be a constructable object.

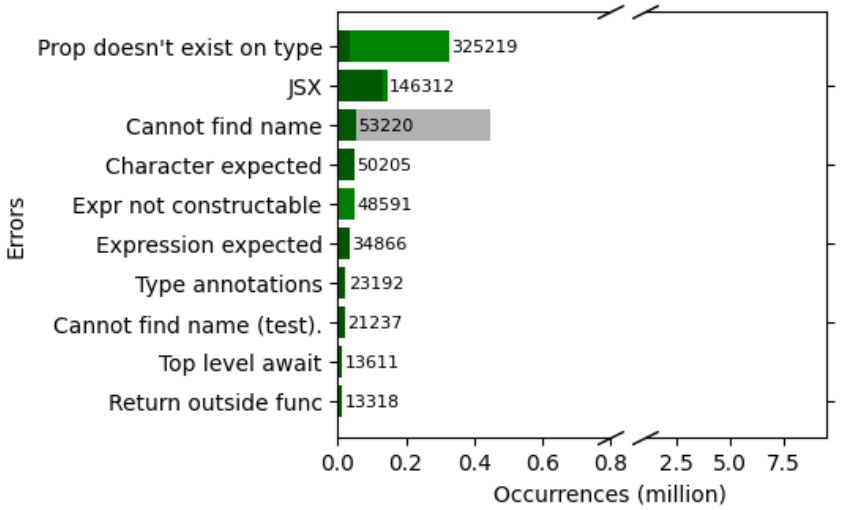

FIGURE 4.11. The 10 most common error types after all fixes. Shading represents prior results from Figure 4.9.

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.

4.5.4 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 4.12 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,

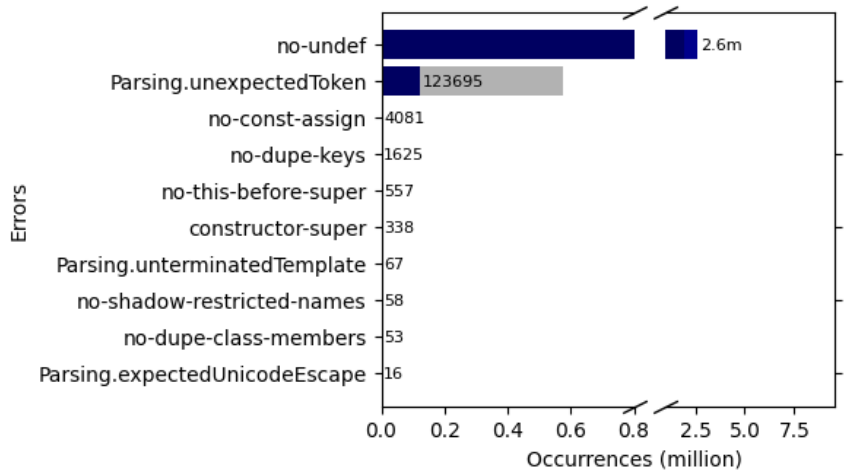

FIGURE 4.12. The 10 most common error types via ESLint after NCQ's corrections. Shading represents the initial values before fixes (see Figure 4.7).

and other non-parsing errors are now visible such as `no-const-assign`. We note from our results that 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.

4.5.5 How does NCC compare to manual fixes?

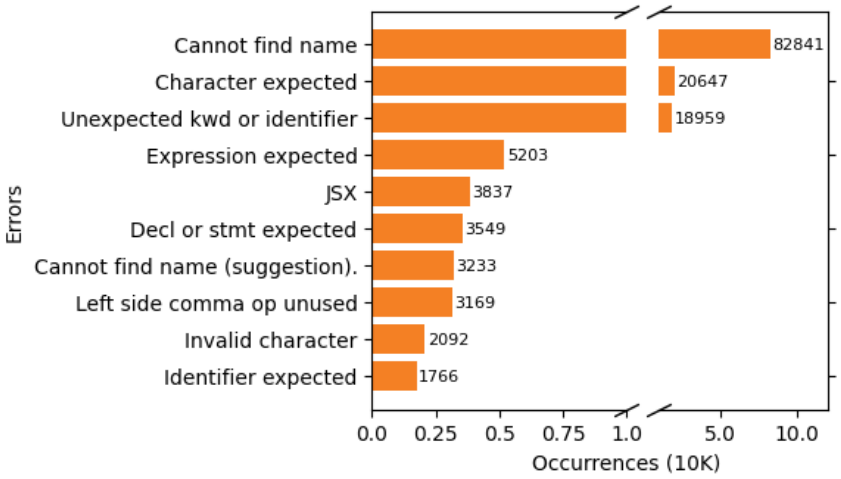

FIGURE 4.13. The 10 most common errors for Stack Overflow Edits pre-edit.

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.

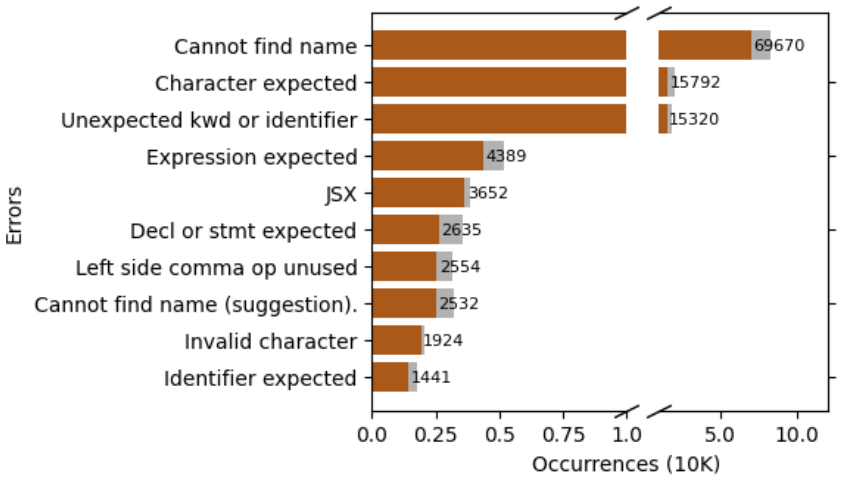

FIGURE 4.14. The 10 most common error types for Stack Overflow Edits post-edit. Grey represents the change between edits.

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 4.13 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.

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 4.14 shows the change in error types, which can be summarised as a general reduction in all common error types.

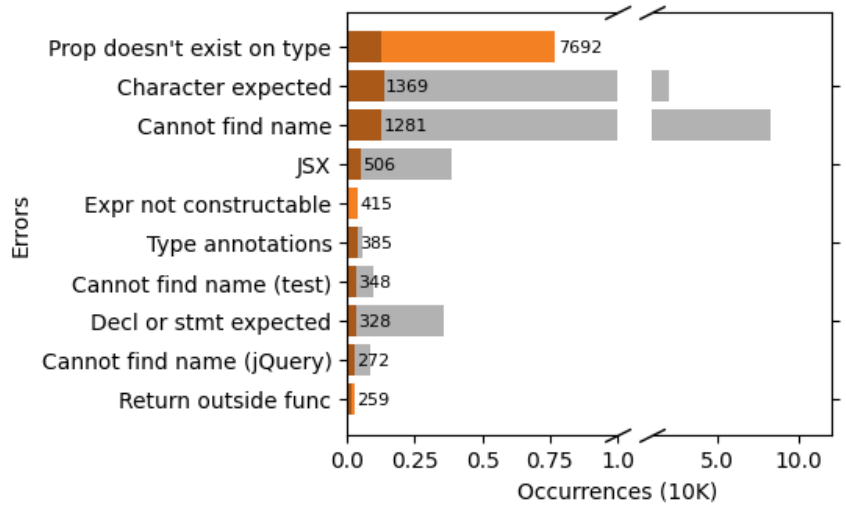

FIGURE 4.15. The most common errors for Stack Overflow snippets after NCC.

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 4.15 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 4.5.3. 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.

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.

4.6 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 4.4, 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.

4.7 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 an 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.

Chapter 5

Generating Code Examples

Related Publication: This chapter is based on our paper: Brittany Reid, Earl Barr, Markus Wagner, and Christoph Treude (2023a). *Copy-Paste vs. Code Generation: Contrasting Stack Overflow and GitHub Copilot*. Being revised for submission to IEEE Transactions on Software Engineering (Reid et al., 2023a).

Large language models, such as Copilot and ChatGPT, are transforming software development by offering alternatives to reuse via online code snippets. Where developers currently copy-paste code examples from online resources like Stack Overflow, tutorials and documentation, that they have to then spend time integrating, they may one day simply generate snippets for their specific situation. To best choose between these code completion approaches, it is essential to understand their strengths and weaknesses. In this study, we systematically compare snippets generated by Copilot and those sourced from Stack Overflow along two dimensions: output quality and user interface. Our findings indicate that Copilot generates at least one snippet per query more frequently than Stack Overflow, but its output is less diverse and prone to hallucinations. Additionally, Copilot snippets tend to contain fewer API calls and exhibit fewer syntax errors compared to Stack Overflow snippets. The effects of query variations, including reducing queries to keywords or incorporating method signatures, differ significantly between the two approaches. These results provide insights into the trade-offs and help software professionals better navigate the rapidly evolving landscape of source code reuse.

5.1 Introduction

Large language models, such as the ones powering GitHub Copilot and ChatGPT, are transforming software development (Lawrence, 2022) and have the potential to replace copy-paste from sources like Stack Overflow as the primary mechanism for code reuse. Over a decade ago, the social media revolution in software engineering (Storey et al., 2010) led to the emergence of platforms like Stack Overflow[1], which centralized copy-paste reuse activity and replaced regular expressions over code with natural language queries and answers that combined code and text. Now, Copilot, ChatGPT, and others are changing this approach once again.

These large language models have the potential to revolutionize the software development process, offering a more sophisticated and efficient way to access relevant code and programming knowledge. This shift effectively reduces the need for manual copy-pasting from Stack Overflow and similar sources. As these models continue to evolve and improve, they may become indispensable tools for developers, ushering in a new era of efficient and intelligent programming practices.

Both copy-paste from Stack Overflow and the use of Copilot and ChatGPT are based on a query-driven programming model, functioning as code completion engines (CCEs). These can be divided into two types: social code completion engines (SCEs), which involve snippet retrieval from crowd-sourced platforms like Stack Overflow, and large language model-based code completion engines (LCEs), such as Copilot and ChatGPT. An SCE query typically consists of natural language search terms (Liu et al., 2021; Cai et al., 2019), while an LCE query is composed of three elements: a cursor (a highlighted code location when the developer issues the query), a code context surrounding the cursor, and a natural language query.

Ultimately, the output of LCEs will define their success. When the output is excellent, developers will adapt their queries and workflows to take advantage of LCEs. Most code is maintained far longer than its initial development, and, to date, code generators have struggled to produce maintainable code (Fry, Landau, and Weimer, 2012), which has hampered their adoption. Will this issue also impact the adoption of code generated by LCEs like Copilot and ChatGPT? Will LCE's tendency to "hallucinate" (Ji et al., 2022) impede their uptake? If the surge in adoption is any indication, early reports suggest it may not (Lawrence, 2022).

To successfully navigate this machine programming wave, the software industry must understand the strengths and weaknesses of LCEs. Answering this question requires clear-headed, empirically rigorous experimentation. In this work, we systematically compare the code provided by LCEs to that available via SCEs along two axes: output quality and user interface.

In terms of the output quality, we find that both CCEs return almost the same number of completions on average: 6 snippets for Copilot *vs.* 6.6. for Stack Overflow, after deduplicating Copilot's completions. This average result masks a difference: Copilot rarely fails to respond to a query, generating at least

[1] https://stackoverflow.com/

one snippet in 99.81% of the cases, whereas Stack Overflow returns nothing in 11.83% of the cases. Stated using averages, this finding understates the difference in the two tool's output. Copilot generates at least one snippet in 99.81% of cases, whereas Stack Overflow returns nothing in 11.83% of the cases. Its snippets are longer on average and contain more duplicate lines. Thus, Copilot rarely fails to respond to query, and yet, despite its loquaciousness, its output is less diverse, on average. If we view completions as draws on a lottery where a winning completion is one that matches a developer's intent, these observations suggest that Copilot's completions are less likely to win as Copilot appears likely to hallucinate some answers and cover less of a query's solution space.

We further find that Stack Overflow snippets contain more comments than Copilot's do: 30.43% *vs.* 7.88%. Copilot is less likely to include working API calls than Stack Overflow: only 3.92% of its snippets contain import statements *vs.* 18.90% of Stack Overflow's. Copilot's snippets are much less likely to contain syntax errors than Stack Overflow's: 10.46% *vs.* 31.43%, a finding that probably understates the ease of using its completions relative to those of Stack Overflow, as Copilot is aware of the local code and more likely to use its variable and conventions. We manually assessed snippet relevance. Our finding here are mixed: Copilot's snippets are more likely to directly addressed the queried problem than Stack Overflow's: 59% to only 6%, but Stack Overflow's are more like to indirectly help: 17% *vs.* Copilot's 9%. Here again, it appears that Copilot is successfully leveraging its immediate awareness of the local context of the completion. These results stand in contrast to the conventional wisdom that Copilot's answers are low quality and Stack Overflow's attendant decision to ban content generated by large language models from its site (Makyen, 2023).

In terms of user interface, both CCEs accept natural language queries, which Copilot supplements with the code context surrounding the cursor when a developer requests a completion. It's likely that both only use a subset of the content of their queries. Guided this hypothesis, we converted their queries to consist only of keywords. Operating on this stripped down queries, Copilot actually produces 55.75% more snippets and that these snippets are more diverse; Stack Overflow returns 44.97% of the snippets it returned on the unmodified query. Clearly, Copilot leverages code context; can Stack Overflow do the same? When we included method signatures in the Stack Overflow queries, it returned 81.50% fewer snippets. In short, it currently does not. How it might do so is an open question.

We use Stack Overflow combined with Google Search as our SCE and GitHub Copilot as our LCE. Since OpenAI will discontinue support for their Codex API[2], we make our dataset of approximately 30,000 code snippets containing almost 500,000 lines of code publicly available[3]. As technology advances rapidly, continuously obtaining up-to-date empirical data is challenging, but it is essential to attempt to understand current trends in order to effectively adapt to and leverage these advancements for improved software development practices. This study aims to generally compare two competing completion models,

[2] https://news.ycombinator.com/item?id=35242069
[3] https://github.com/ctreude/stack-pilot

the earlier query-based copy and paste and the newer generative model, using Stack Overflow and Codex as our exemplars. We have designed this study with this goal in mind: to capture the intrinsic differences between these models, not accidental differences due to our choice of exemplars.

5.2 Research Questions

We ask a series of research questions about the dataset, split into themes: **Output Quality** and **User Interface**.

The following research questions concern the differences in **output quality** between Copilot and Stack Overflow. Existing work has identified quality concerns with the output of both large language model-based code completion engines and online code reuse (Al Madi, 2022; Wong, Kothig, and Lam, 2022; Erhabor, 2022; Yang, Hussain, and Lopes, 2016; Meldrum et al., 2020), however, there is a lack of comparison between the two. We attempt to characterize the differences, in order to gain a better understanding of how AI generated code differs from human-written code, and if it can adequately serve developers needs for code snippets in its current form. We hypothesize that AI code generators like Copilot will return output for all queries, but that due to this, the output may not always be of value. To measure output quality, we look at the properties of the code, as well as how the number of returned snippets varies, and the variation between snippets for the same query using Normalized Compression Distance (NCD) (Yu, Lee, and Yu, 2021), which we introduce in more detail in Section 5.4.2.

First, we investigate the differences between each CCE, asking:

RQ1.1 How many snippets does each CCE return?
RQ1.2 How do different snippets for the same query differ?

Next, we seek to uncover software engineering salient properties of the code snippets, by looking at snippet length, comments, API usage and errors, asking:

RQ2.1 How do snippets differ in length?
RQ2.2 To what extent do snippets returned by the approaches contain natural language explanations?
RQ2.3 How does machine-generated code differ from human-written code in terms of API usage?
RQ2.4 What errors do the snippets have?
RQ2.5 How relevant are snippets?

The following research questions relate to the **user interface**, i.e., the way that users interact with the CCEs. Currently, developers interact with code search systems using natural language queries, which we observe the results of in the preceding section. However, underlying both systems is keywords for traditional web search and the code context for Copilot, where queries are simply embedded as comments. We hypothesize that changing the type of input will impact results. Additionally, we expect that Copilot will perform better than Stack Overflow at understanding code. Similarly, if Large Language Models

are meant to handle code and natural language, they may perform worse with keywords. We modify our set of queries and observe the differences in output:

RQ3.1 How sensitive are the approaches to the use of complete sentences vs keywords for queries?
RQ3.2 How sensitive are the approaches to inclusion vs exclusion of method signatures?

5.3 Dataset

TABLE 5.1. Overview of queries. † Unlike the other sources, soSearches uses distinct queries for Java and Python.

Source	Queries
humanEval	324
leetcode	768
leetcodeTitles	768
soSearches	384†

We compile a set of natural language queries for both Python and Java from multiple sources: popular search queries from Stack Overflow, programming tasks from LeetCode[4] and problems from HumanEval (Chen et al., 2021), an evaluation set for code synthesis. In total, we mined 1,698 unique queries, a set of 1,314 for each language. This gives us a combined set of 2,628 queries.

Table 5.1 illustrates the breakdown of queries by source. We define four sources of queries: *soSearches*, *humanEval*, *leetcode* and *leetcodeTitles*. For *soSearches*, we took 384 of the most popular Stack Overflow search queries for both Java and Python, for a total of 768 queries. We sourced the 162 *humanEval* queries by taking the docstring description from the prompt of each entry, and the 384 *leetcode* queries from a random sample of LeetCode problem descriptions. We limited each of these to the first 32 words (the limit for Google search queries). For *leetcodeTitles*, we instead used the problem titles for the same set. The *humanEval*, *leetcode* and *leetcodeTitles* queries were not language specific, so were used for both Java and Python sets.

Using this set of queries, we mined both Stack Overflow and Copilot for snippets. We consider both sources of code snippets to be **Code Completion Engines** (CCEs). Each CCE was given an input, a **query**, and returned output in the form of **code snippets**. While Stack Overflow returns more than just code, including natural language explanations, for the purposes of this comparison and for defining Stack Overflow as a CCE, we only make use of the code snippets embedded within answers. Software Developers often copy-and-paste code from Stack Overflow, and like Copilot, there exist similar extensions that embed Stack Overflow snippets within the IDE (Ponzanelli, Bacchelli, and Lanza, 2013; Campbell and Treude, 2017; Reid, Treude, and Wagner, 2020).

[4] https://leetcode.com/

Copilot generates snippets for a code context and cursor location, where natural language queries are embedded within the code as comments. Figure 5.1 demonstrates how we provided a query as a comment above the beginning of an incomplete function declaration in both languages, where we triggered the Copilot extension at the end of the code to generate a completion of the function. Copilot generates a maximum of 10 completions per query.

(A) Python input

```
1  # {query
     }
2  def |
```

(B) Java input

```
1  // {query}
2  public class Clazz
     {
3    public |
```

FIGURE 5.1. Python (a) and Java (b) input given to Copilot. A vertical line (|) indicates cursor position when the extension was evoked.

To programmatically mine our set of 2,628 queries, we created a VSCode extension that inserts the input code into an empty file, sets the VSCode language, calls the Copilot extension, waits 20 seconds for code to be generated, then saves the output from the Copilot window. The 20-second timeout was selected based on experimentation, as it takes differing amounts of time to generate all 10 results. Our extension is available on GitHub[5]

We mined **Stack Overflow** using Google to search for queries. This process mimics how developers might search for code; using a general search engine to direct them to answers. Using SerpApi[6] to retrieve Google results, we performed a site specific search for each query. For each page of Google results, we retrieved a list of Stack Overflow question URLs. Then, we used the SO API to get only the accepted answer for these questions. We consider the 'code snippet' to be the first section of code in the text, by looking for the first `<code>` HTML formatting. We continue retrieving Google results until we have 10 snippets (matching Copilot's default) or there were no results left. The order of snippets thus reflects Google's own ordering.

5.4 Results

5.4.1 RQ1.1: Number of Snippets Returned

The number of code snippets for each query is of interest due to the cognitive load this places on developers. That is, developers may perform worse when

[5] https://github.com/Brittany-Reid/copilotmine
[6] https://serpapi.com/

presented with more choices that they need to evaluate (Cockburn, Gutwin, and Greenberg, 2007). On the other hand, more choice provides alternative solutions to a task. We are particularly interested in the number of results returned per query, and cases where no results are returned. Due to the nature of AI code generation, we expect Copilot to return results in all cases.

TABLE 5.2. Overview of the Dataset

	Queries	Snippets	
		SO	CP
Java	1,314	7,897	7,843
Python	1,314	9,478	7,889
Total	2,628	17,375	15,732

In Section 5.3 we describe how we designed the dataset generation to be comparable between CCEs, by attempting to generate 10 code snippets for each query. Despite this, the number generated in total still differs between CCEs, as can be seen in Table 5.2. This overview of the dataset reveals that contrary to our assumptions, Copilot actually generates fewer snippets (an average of 6 snippets to Stack Overflow's 6.6), and in total Stack Overflow returns 1.104 times the number of snippets as Copilot. The distributions of snippets per query are shown in Figure 5.2. There are also an additional 10 cases (0.38% of queries), not shown, where Copilot behaves unexpectedly and generated more than 10 snippets (up to 14).

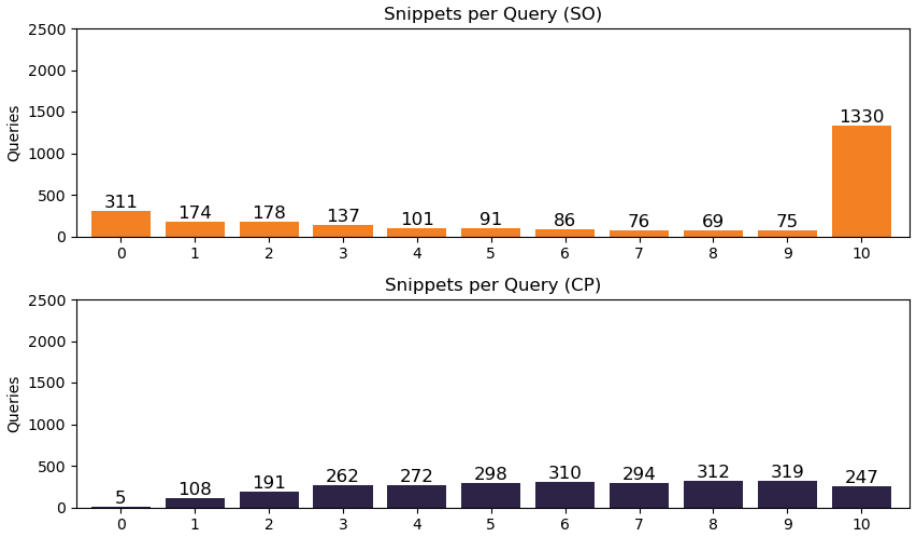

FIGURE 5.2. Distribution of number of snippets per query, for each CCE. Not shown: 10 cases where Copilot generated more than 10 snippets.

However, it is important to note that Copilot automatically filters duplicate snippets from its results. Because Copilot reports the original number of results, we can compare this to the returned results to calculate the number that were

filtered. We found that for 84.93% of queries, Copilot removed duplicates, for a total of 9,642 duplicates removed (and without removal the total set would have 25,374 snippets, an average of 9.6 snippets per query). 2,325 queries (88.47%) return 10 snippets. Additionally, when we checked the 17,375 Stack Overflow code snippets, only 470 were duplicates of another snippet for the same query. Thus, it is not fair to say that Copilot generates less code, but that it generates less *diverse* code.

On occasion, queries return no snippets. This occurred 311 times in the Stack Overflow set, but only 5 times in the Copilot set. There was no overlap between the set of non-returning queries between CCEs. As described in Section 5.3, we reran failed cases such as these to account for internet connectivity issues during the mining process. Of the 5 Copilot queries, all lacked important information for the solution (for example, the query "Minimum Limit of Balls in a Bag"); however, other similar queries did generate results, so we do not know why these cases fail.

Finding 1.1: Number of Snippets Returned

After deduplicating Copilot's completions, both Stack Overflow and Copilot return almost the same number of snippets per query on average: 6 snippets for Copilot *vs.* 6.6 for Stack Overflow. This average result masks a difference: Copilot rarely fails to respond to a query, generating at least one snippet in 99.81% of the cases, whereas Stack Overflow returns nothing in 11.83% of the cases.

The finding that the average number of snippets returned overturned our initial, subjective impression of using the two reuse techniques: that Copilot offered many more suggestions. Perhaps our impression was due to the fact that we found it easier to navigate Stack Overflow's threaded results. Silence can be golden, especially when the alternative is distracting. So, while we were surprised to discover that Copilot and Stack Overflow generate roughly the same number of snippets to consider, Copilot's tendency to be overconfident and hallucinate shows up here in long tail of shorter snippet lists.

5.4.2 RQ1.2: Snippet Diversity

We are interested in the diversity of snippets per query; that is, "When there is more than one snippet, how different are the options given?". The raw number of snippets may not necessarily be a benefit; additional options are only useful if they *are different enough*. For example, if the first snippet for a query fails to solve the task, there is a possibility the next might be an option. We conjecture that there will be more variation between snippets in code written by humans; in contrast, we hypothesize that Copilot will generate more similar responses per query; already we observe that its built-in duplicate remover triggers on 84.93% of queries.

We measure similarity of snippets for the same query using Normalized Compression Distance (NCD), which measures the similarity between two objects, based on the difficulty to compress the two objects combined compared to on their own. We use LZMA to compress our snippet strings, because it satisfies

normality properties better than other compression algorithms (Morgan, 2015; Cilibrasi and Vitányi, 2005). For all pairs of snippets (x, y) in a single query, we calculate the NCD of each pair as $min(NCD(x, y), NCD(y, x))$, as this is a closer approximation of the true NCD (Yu, Lee, and Yu, 2021).

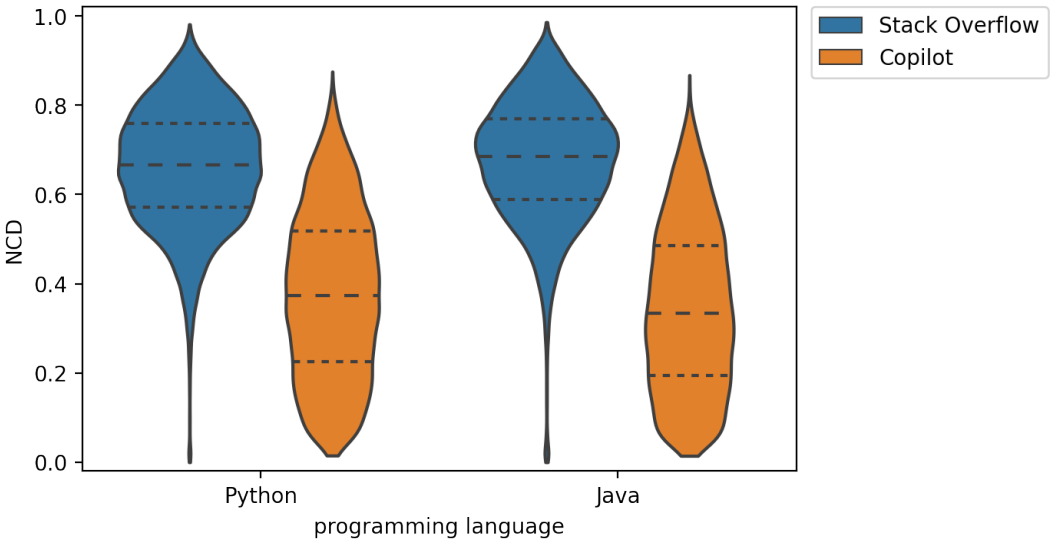

FIGURE 5.3. Distribution of NCDs for each CCE and language.

Figure 5.3 shows the distributions of NCDs for each language and CCE, where a higher NCD indicates less similarity between two snippets from the same query. For the Stack Overflow Python set, the mean and median NCDs were 0.660 and 0.667; for Java these were 0.674 and 0.684. For Copilot, Python values were 0.376 and 0.373; for Java 0.344 and 0.333. In general, the average NCD between same-query snippet pairs was higher for Stack Overflow than Copilot, indicating more diversity in answers. The maximum NCD in the sets were (Stack Overflow Python) 0.981, (Stack Overflow Java) 0.984, (Copilot Python) 0.875 and (Copilot Java) 0.867, while the minimums were 0 for Stack Overflow and 0.015 (Python) and 0.014 (Java) for Copilot. The lack of 0 NCD in the Copilot set reflects the pre-filtering. These values indicate that the range of diversity is larger in the Stack Overflow set. For each language, Cohen's d characterizes the differences between the two CCEs as 'huge'.

Finding 1.2: Snippet Diversity

Using NCD, we find that Stack Overflow snippets are more diverse per query for both languages. Average NCDs for Java snippets were 0.674 and 0.334 for Stack Overflow and Copilot respectively.

These findings indicate that on a per-query basis, returned snippets have more diversity from Stack Overflow than from Copilot. That is, our assumption that Copilot's generated snippets are more similar to each other is confirmed. Alternative snippets may be less likely to be of use in Copilot; in situations where developers would be looking beyond the first snippet finding only similar snippets may be less than useful. Contrastingly, Stack Overflow is likely to suggest a more diverse range of snippets that could be of more help.

5.4.3 RQ2.1: Snippet Length

Length is an important characteristic of code; the longer the code, the more errors it is likely to have (Lipow, 1982). Additionally, developers may be able to comprehend shorter code snippets faster than more verbose ones. This question is also useful for characterizing the type of code that each CCE outputs. We calculate lines of code (LOC) for each snippet by simply counting non-empty lines. The disadvantage of this method is that it includes comments, and statements split across multiple lines. The advantage is that because the dataset includes erroneous snippets (which we discuss in Section 5.4.6), the count is consistent regardless of parsing errors; when using tree-sitter to count statements, some snippets report as empty.

TABLE 5.3. Lines of code per CCE and Language

	SO	CP	Total
Total	199,086	275,770	474,856
Java	108,752	146,233	254,985
Python	90,334	129,537	219,871

Table 5.3 shows the total LOC in our dataset by CCE and Language. In total, the dataset consists of 474,856 lines of code. We observe that Copilot generates 38.52% more lines for the same set of queries than Stack Overflow. The mean code snippet length was 11.46 lines for Stack Overflow, and 17.53 for Copilot. All snippets consist of at least one line, with a maximum length of 496 lines for Stack Overflow and 138 lines for Copilot. Java snippets were longer on average than Python for both CCEs, but Copilot snippets being longer on average remained consistent between languages (18.65 lines vs 13.77 for Java, 16.42 vs 9.53 for Python).

```
1  def count_numbers_with_unique_digits(n):
2    if n == 0:
3      return 1
   ...
18   if n == 8:
19     return 3579139
20   if n == 9:
21     return 14893690
   ...
79   if n == 8:
80     return 2613122
81   if n == 9:
82     return 7657651
   ...
```

FIGURE 5.4. Truncated version of the largest Copilot snippet, demonstrating repetition in Copilot set.

Upon analysis of the largest snippets, we saw that the longest Stack Overflow snippet was an entire program implementing a representation of Fractions (Kip,

2009), while the Copilot snippet was a nonsensical repetition of statements. A snippet of this code, for the query 'Count Numbers with Unique Digits', is shown in Figure 5.4. Other snippets for this query in python followed a similar pattern. Throughout the Copilot dataset, we notice this type of duplication of statements. This may account for the higher line count on average.

To see the extent of this code repetition, we count the number of duplicate lines per snippets in each set. We expect a level of repetition within code snippets; for example, the most repeated line in the Stack Overflow set was '`}`', a closing bracket used in Java. However, for the Copilot set, the most common repeated line was '`public`'. Upon closer look, we saw that the affected code snippet was a series of 59 strings being declared with the '`public`' on a separate line. On average, Copilot snippets contained 5.35 repeating lines, compared to Stack Overflow's 1.95. 27.75% of Stack Overflow code snippets contained repetitions compared to 66.85% of Copilot code snippets.

Finding 2.1: Snippet Length Differences

Copilot generates 38.52% more lines of code than Stack Overflow, so its snippets are longer on average. They also have more duplicate lines.

In general, Copilot generates more code for the same set of queries, the incidence of repetition within its snippets again points to the lack of diversity. If the repetition is characterized by the type of snippets displayed in Figure 5.4, this indicates Copilot may generate a sizable amount of 'junk' responses. Additionally, because of the propensity for longer snippets to have more bugs, Copilot may be inferior in this regard.

5.4.4 RQ2.2: Comments in Snippets

We look at the frequency of comments within snippets generated by both CCEs. Comments can provide documentation or explanations of code, and can make comprehension of code easier; for example, code on Stack Overflow is often accompanied by a natural language explanation. We used tree-sitter to identify comments in code.

Table 5.4. Comments per CCE and Language

	SO	CP	Total
Total	19,282	11,618	30,900
Java	8,803	252	9,055
Python	10,479	11,366	21,845

Table 5.4 shows the total number of comments per CCE and Language. The Stack Overflow set contains 65.97% more comments than the Copilot set; 30.43% of Stack Overflow code snippets include comments, while only 7.88% of the Copilot set do. The average number of comments per snippet were 1.11 and 0.74 respectively. However, when we exclude the no comment set, we observe

```
def main():
  # create a list of items
  items = [
      "apple",
      "banana",
      "grapes",
      "orange",
      "pineapple"
  ]
  # display the list
  print(items)
  # display the list
  print(items[0])
  # display the list
  print(items[1])$\lstsetnumber{}$
  ...$\lstresetnumber\setcounter{lstnumber}{68}$
  print(items[31])
  # display the list
```

FIGURE 5.5. Example of repeating comments in the Copilot set.

that when there *are* comments, Copilot snippets have more on average (9.38 comments vs Stack Overflow's 3.65).

Because repetition was observed in the LOC analysis (Section 5.4.3), we again measure the number of duplicate comments per snippet. Due to the nature of comments, we expect to see some repetition; for example, the comment 'TODO' appears 9 times in a single snippet. 32.1% of Copilot snippets with comments included duplicates, compared to just 5.8% of Stack Overflow snippets. The highest duplicate count for a comment per snippet in the Stack Overflow set was 18, where the same comment was repeated for a series of automatically generated method declarations. For Copilot, the most repeated comment was 'display the list" at 34 instances; this code snippet is shown in Figure 5.5 and appears to show a similar pattern of nonsensical repetition as was observed in Figure 5.4.

Finding 4: Comments in Snippets

Stack Overflow snippets contain 30.43% more comments than Copilot's at 7.88%. When snippets *do* include comments, Copilot snippets have more per snippet: 9.38 *vs.* Stack Overflow's 3.65. Additionally, Copilot has a higher rate of duplicate comments 32.1% *vs.* Stack Overflow's 5.8%.

Stack Overflow snippets appear more likely to include comments, perhaps as a consequence of the 'answer' aspect of the website. Developers answering questions may feel the need to explain parts of their code, which might not otherwise be commented in project code. This suggests Stack Overflow code snippets are demonstrational and explanatory in a way Copilot snippets are not.

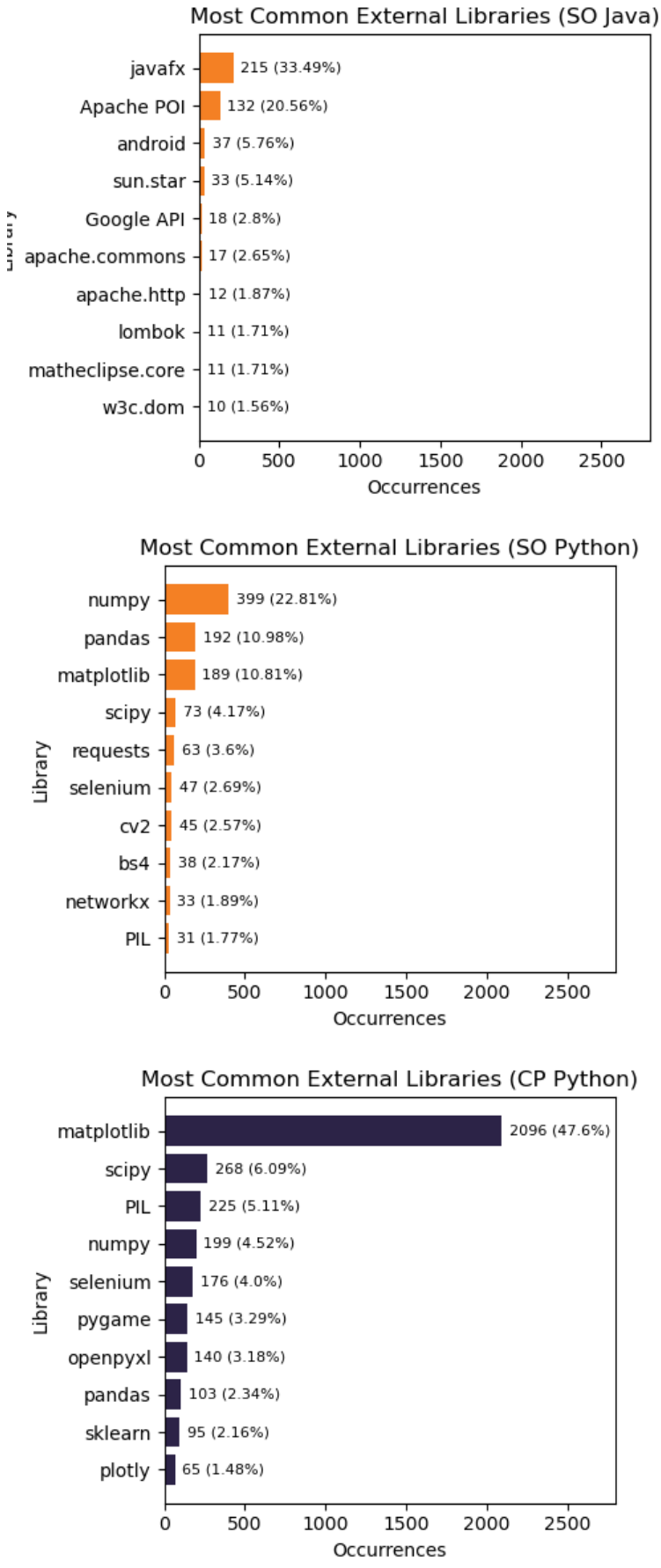

FIGURE 5.6. Most common external libraries for each CCE and language. Note that Copilot has no imports for Java.

5.4.5 RQ2.3: Snippet API Usage

Developers commonly use third-party libraries to reduce the amount of code they need to write and maintain. With this in mind, we ask if API usage differs between machine and human-written code. We hypothesize that Copilot is more likely to avoid API usage; unlike the human developers answering Stack Overflow questions, it has no understanding of what libraries exist, where to find libraries, when a library can be useful, and how libraries benefit development. This would suggest that, ironically, a tool for speeding up code-writing such as Copilot might actually be increasing technical debt and riddling a code base with micro-code clones (or near clones).

Table 5.5. Import statements per CCE and Language

	SO	CP	Total
Total	6,496	8,313	14,809
Java	2,591	0	2,591
Python	3,905	8,313	12,218

First, we analyse the frequency of import statements within the dataset. Table 5.5 shows the total number of import statements for Stack Overflow and Copilot, with additional language breakdowns. Additionally, we observe the number of snippets with at least one import statement: only 617 of Copilot code snippets contain an import statement compared to 3,283 Stack Overflow snippets (3.92% and 18.90% of all snippets respectively). Per language, we see that Copilot generates no Java snippets with import statements. This can likely be explained by the evocation location; we ask Copilot to complete a Java function in a class, and import statements can only be declared at the top of a file unlike in Python.

We find that the mean number of imports within the set of snippets with at least one import was 1.98 for Stack Overflow and 13.47 for Copilot. While the median number was 1 and 2 imports, the maximum per snippet was 32 and 112, respectively. Again, we checked for repetition to explain these outliers. We observe a similar pattern as in previous sections: 21.72% of Copilot code snippets with imports had duplicates, compared to just 4 cases in the Stack Overflow set. For these 4 cases, the duplication was a single instance, while the average duplicate for the 134 Copilot cases was 27.40, indicating an extreme amount of duplication for this aspect of code.

Next, we characterize the use of third-party libraries in particular. External libraries, compared to standard libraries like Java's `ArrayList` and Python's `math`, made up 36.81% of Stack Overflow and 52.97% of Copilot import statements. 7.01% of Stack Overflow and 2.36% of Copilot snippets imported an external library. Figure 5.6 shows the top 10 third-party libraries for each dataset (note that Copilot has no Java imports), and the percent of external imports each accounts for. 37.60% of external imports in the Copilot set belong to the `matplotlib` library. We observe similar packages within the most common subset across both CCEs: `numpy`, `pandas`, `scipy`. Again, we observe less

diversity in the Copilot set: there were a total of 184 unique external libraries imported compared to Stack Overflow's 242.

Additionally, we looked at method calls, as import statements may not describe the entire extent of API usage within snippets. Two authors manually annotated if method calls likely belonged to a third-party library or not, looking at the entire method call (for example, the method `plt.subplot()`) likely belongs to the python library `matplotlib`). For ambiguous cases, we defaulted to the majority case (not external). Authors annotated a sample of 739 method calls; this sample represents all method calls for each CCE and language with 20 or more occurrences within the dataset: 377 for Copilot (202 Java and 175 Python) and 362 for Stack Overflow (176 Java and 186 Python). Authors first both annotated the same random sample of 50; for a Cohen's Kappa of 0.65 (substantial agreement). After discussing inconsistencies, both authors then independently rated half of the remaining set each.

```
1   static void main(String[] args){
2     WebDriver driver = new FirefoxDriver();
3     driver.get("http://www.google.com");
4     driver.findElement(By.name("q")).sendKeys("Selenium");
5     driver.findElement(By.name("btnG")).click();
6     driver.close();
7   }
8 }
```

FIGURE 5.7. Copilot Java Snippet using the Selenium API.

Our method call analysis reveals that 12.47% of Copilot method calls were for external libraries, compared to Stack Overflow's 17.13%. We also observed that while Copilot Java snippets did not include import statements, 9.41% of method calls still made use of third-party libraries such as Selenium and Apache Commons. Figure 5.7 shows one such code snippet for the method call `driver.get()`.

Finding 5: Snippet API Usage

Copilot is less likely to use APIs: only 3.92% of snippets import a library. Additionally, 7.01% of Stack Overflow and 2.36% of Copilot snippets imported a third-party library. Despite no Java snippet importing a library, Copilot snippets still make use of API calls.

Because Copilot is less likely to use APIs, it may result in snippets that add more technical debt to a project where library use is preferable. Additionally, because Copilot still uses APIs without import statements, as demonstrated in the Java subset, developers need to spend time inferring what libraries a snippet makes use of.

5.4.6 RQ2.4: Snippet Errors

Code snippets sourced from online are often fragmented and, when copy-and-pasted, contain errors that make code reuse difficult (Yang, Hussain, and Lopes,

2016); developers then devote time to fixing these errors and correctly integrating code into their projects. We ask if Copilots additional code context improves the rate of errors, or, as discussed in previous sections, if it is more likely to generate 'nonsense' snippets.

Table 5.6. Errors per CCE and Language

	SO	CP	Total
Total	22,903	2,938	25,841
Java	13,669	1,660	15,329
Python	9,234	1,278	10,512

To gain an understanding of what problems snippets have, we analyse errors reported by tree-sitter. Tree-sitter reports erroneous and missing nodes when it encounters issues parsing code, but it does not report any other error information. In addition, parsing errors do not reveal the full scope of errors; parsable code may still not run. For this analysis, we parsed Copilot snippets with the generation context attached; that is, Java snippets begin after the `public` shown in Figure 5.1. Without this change, all Copilot Java snippets would generate errors. Python generations include the supplied `def` already, so needed no change. For Stack Overflow snippets, we recreate a similar realistic copy-and-paste scenario for Java; we parse snippets as if they are pasted into an empty class and main function. Python is less strict about structure, so we make no changes for this set.

Table 5.6 reports the number of errors per CCE and Language, where Stack Overflow snippets have 679.54% more errors than Copilot snippets. Tree-sitter reports that 31.43% of Stack Overflow snippets and 10.46% of Copilot snippets have errors. On average, Stack Overflow snippets have 1.32 errors, while Copilot snippets have 0.19. This trend continues per language: 45.71% of Java Stack Overflow snippets have errors vs. 12.07% of Copilot's. For Python this is 19.53% vs. 8.86%.

Finding 6: Snippet Errors

Copilot snippets are not error free; however, we find that Stack Overflow snippets are more likely to have parsing errors; 31.43% of Stack Overflow *vs.* 10.46% of Copilot snippets.

This error analysis suggests that the code context provided with Copilot queries does enable Copilot to better integrate snippets within an existing structure, thus reducing integration work required of traditional copy-and-paste code reuse. However, Copilot snippets are not free of errors, and may still require developers to spend time fixing errors; it still appears to be significantly less than with snippets from Stack Overflow.

5.4.7 RQ2.5: Snippet Relevance

To evaluate the relevance of code snippets, we annotated a subset of the data based on their relevancy in addressing the initial query. We randomly selected

336 queries to achieve a 95% confidence level with a 5% margin of error. The rating scale we used to measure relevance is a variation of the one proposed and used by Mahajan, Abolhassani, and Prasad (2020), which was originally based on Begel and Zimmermann (2014), Lo, Nagappan, and Zimmermann (2015), and Kitchenham and Pfleeger (2008). Mahajan et al. employed four distinct categories to measure Stack Overflow post relevancy:"Instrumental", "Helpful", "Misleading", and "Unavailable".

Since our project focuses on code snippets rather than the availability of results, we excluded the "Unavailable" category and added a "Relevant Post" category to our rating scale. We rated Stack Overflow results as "Relevant Post" if the code snippet was "Misleading", but the post itself could be considered "Instrumental" or "Helpful".

We labeled a code snippet as "Instrumental" when it perfectly addressed the initial query, and conversely, we labeled it as "Misleading" when it provided no help in answering the query. Between these extremes are "Helpful" code snippets. We further classified the "Helpful" category into six reasons for assigning this label.

TABLE 5.7. Relevance of code snippets to query. he = humanEval, lc = leetcode, lt = leetcodeTitles, so = soSearches

Rating	he	lc	lt	so	Total
CP					
Instrumental	59%	44%	45%	53%	**169 (50%)**
Helpful	9%	9%	11%	7%	**28 (8%)**
Misleading	31%	47%	44%	41%	**139 (41%)**
Stack Overflow					
Instrumental	6%	9%	29%	14%	**48 (14%)**
Helpful	17%	8%	15%	26%	**60 (18%)**
Relevant Post	19%	13%	12%	15%	**49 (15%)**
Misleading	59%	71%	44%	45%	**179 (53%)**

To mitigate the subjectivity of the annotation process, we implemented several steps beyond simply having the annotator read the code. For Stack Overflow code snippets, we also reviewed the original post from which the snippet was taken, taking into account other users' comments. For Copilot code snippets, we employed a variety of methods for annotating, including running and testing the code on a local IDE, consulting ChatGPT, and for LeetCode queries where the task was clear, submitting the code as an answer to the corresponding LeetCode problem and noting its acceptance.

As shown in Table 5.7, when comparing the relevance of code snippets, Copilot performed better overall, with a higher percentage of code snippets being "Instrumental" or "Helpful" (58% vs. 32%) and a lower percentage of "Misleading" code snippets (41% vs. 53%). Furthermore, when Copilot-generated code snippets were useful, they were significantly more likely to be "Instrumental"

rather than "Helpful" (50% vs. 8%), in contrast to Stack Overflow code snippets (14% vs. 18%).

As illustrated in Table 5.8, focusing on why Stack Overflow code snippets are more likely to be labelled as "Helpful" instead of fully "Instrumental", the most prominent reasons include the code snippet solving a more complex problem than the task required, with the solution itself still contained within (35%), or the code snippet helping with part(s) of the task but not solving it as a whole (32%). These were also the top two reasons for "Helpful" Copilo code ratings, albeit at different percentages of 21% and 50%, respectively.

Another observation was that there were cases where the code snippet provided from Stack Overflow was correct for a different circumstance but still helpful overall; for example, by providing the code to draw a frowny face instead of a smiley face as queried. These scenarios were not seen in any of the annotated Copilo code snippets. When Copilo code snippets were "Helpful" or "Instrumental", they more often targeted the specific query circumstances and frequently included suitable method naming for the task.

Finding 7: Snippet Relevance

Copilot produced more instrumental or helpful code snippets and fewer misleading ones compared to Stack Overflow. Stack Overflow snippets often encountered scope-related issues, with the scope being either too large or too small, which hindered them from perfectly addressing the query.

The challenges associated with code reuse are far from being resolved, as both CCEs continue to produce misleading code snippets in many instances. Nevertheless, large language models appear to have tackled the issue of mismatch between query scope and solution scope, which is frequently observed on Stack Overflow.

5.4.8 RQ3.1 Keyword Queries

Traditional search engines, like the Google search used to generate our Stack Overflow dataset, accept natural language queries. However, search engine indexing and retrieval operates on the keywords within those queries. Search engines also often ignore common words that don't contain topical information — stop words like 'the' and 'a'. Similarly, Copilot accepts a code context which can have natural language embedded within as comments, as we do for our queries. We ask then, what happens when we modify queries to have only keywords? Does the lack of syntactic information affect the results of LLMs like Copilot? How different are the results with this information removed? We expect Stack Overflow to be less affected by the change, but the reduction in information provided to Copilo may enable less specific results.

We take our initial set of 2,628 queries, and utilize `nltk` (Project, 2023) to generate keyword-only queries, using part-of-speech tagging to only retain nouns, adjectives and verbs. For example, the query 'how to parse .csv file in python' becomes 'parse .csv file python'. We then repeat the same mining process used in Section 5.3 to generate a second set of snippets from both Stack Overflow and Copilot.

TABLE 5.8. Further breakdown of "Helpful" results. he = humanEval, lc = leetcode, lt = leetcodeTitles, so = soSearches

Reason	he	lc	lt	so	Total
CP					
Code does not run as intended but generally has the right approach		7%			**2 (7%)**
Code snippet solves a more complex problem than query (but solution is contained within)				21%	**6 (21%)**
Code is correct but may lead to errors in some likely to occur instances	11%	7%			**5 (18%)**
Code helps with part(s) of query but does not solve the task as a whole	7%	11%	25%	7%	**14 (50%)**
Code is incomplete or missing key parts				4%	**1 (4%)**
SO					
Code does not run as intended but generally has the right approach	2%				**1 (2%)**
Code snippet solves a more complex problem than query (but solution is contained within)			5%	30%	**21 (35%)**
Code is correct but for a different circumstance (overall still helpful)	2%		2%	3%	**4 (7%)**
Code is correct but may lead to errors in some likely to occur instances	2%	2%	2%	10%	**9 (15%)**
Code helps with part(s) of query but does not solve the task as a whole	8%	7%	7%	10%	**19 (32%)**
Code is incomplete or missing key parts	2%	2%	2%	5%	**6 (10%)**

TABLE 5.9. Overview of the Keyword Query Dataset

	Queries	Snippets		
		SO	CP	Total
Total	2,628	17,133	24,502	41,635
Java	1,314	7,900	12,180	20,080
Python	1,314	9,233	12,322	21,555

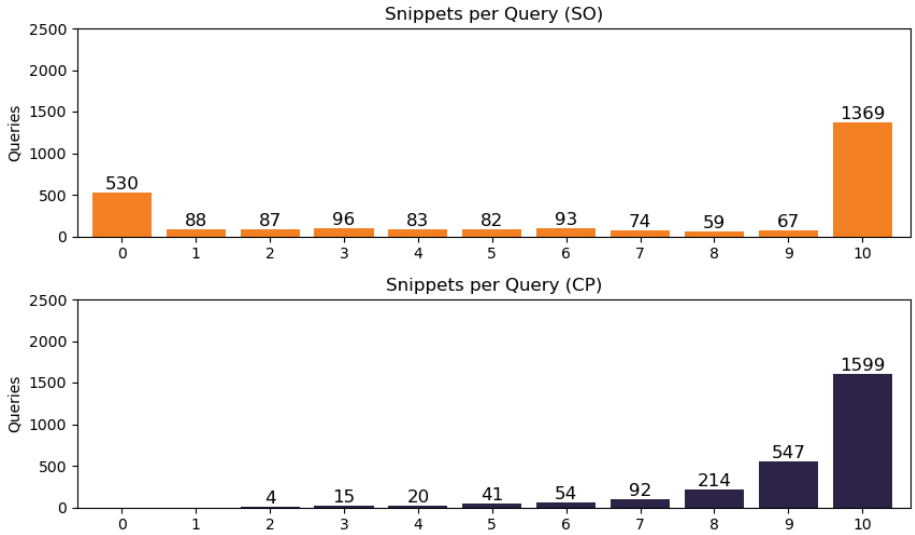

FIGURE 5.8. Distribution of number of snippets per **keyword** query, for each CCE.

Table 5.9 shows the resulting dataset. We see that the number of Stack Overflow snippets remains similar between query variations (a 1.39% decrease in snippets), but that the Copilot set has increased from 15,732 snippets to 24,502 (a 55.75% increase). All Copilot queries return at least 1 snippet, however, Stack Overflow fails to generate results for an additional 219 queries, for a total of 530 queries. Figure 5.8 shows the distribution of number of snippets per query: on average Copilot generates 9.32 snippets per query, to Stack Overflow's 6.52. Copilot's built-in duplicate removal only triggers for 567 queries now (21.58%), to remove 1,223 snippets. When we look at exact matches between the returned snippets before and after query modification, we observe that 7,704 Stack Overflow snippets are returned for the same queries in both sets (44.97%). However, only 77 snippets in the Copilot set are exact matches. 1,782 (67.81%) Stack Overflow queries return at least one of the same snippets regardless of keyword usage. The same can only be said for 69 Copilot queries.

Again, we look at the snippet diversity using NCD. We find that the average NCD between snippets for the same query remains similar for Stack Overflow: an average NCD of 0.660 for Java and 0.674 for Python. However, Copilot average NCD increases from 0.344 and 0.376 to 0.429 and 0.507 for Java and Python respectively, indicating a greater degree of difference between snippets returned for the same query. Figure 5.9 shows the distribution of NCDs for

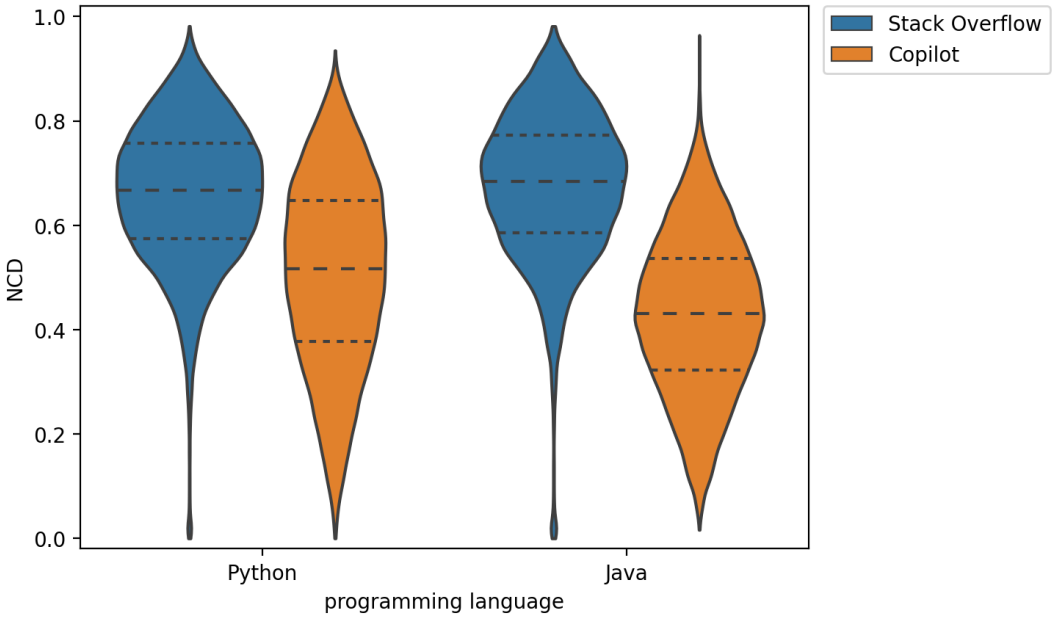

FIGURE 5.9. Keyword query distribution of NCDs for each CCE and language.

same-query snippet pairs.

Finding 8: Keyword

Stack Overflow results remain similar with the use of keyword queries, but Copilot generates 55.75% more snippets, filters duplicates in fewer cases, and returns more diverse snippets. Additionally, 44.97% of Stack Overflow snippets are returned for both query types.

The Copilot results indicate that the shortened, keyword queries enable Copilot to generate more, and more diverse, snippets. However, over half of Stack Overflow's snippets are different. Replacing natural language queries with keywords impacts the output of both CCEs. Keyword only queries remove information from queries, possibly resulting in more ambiguous queries; the increase in diversity may also result in a decrease in relevancy.

5.4.9 RQ3.2 Method Signature Queries

Copilot deals with code context to generate its completions, and so we ask: what happens when we include code with our queries? Software development already takes place within the domain of code; search-based code reuse involves an additional step of taking an idea or problem and encoding it in natural language to form a query. Thus, the interface of code offers a new way to reuse code that might be more intuitive to developers. We expect that Copilot will perform preferably here, while traditional, general-purpose search engines are not built to understand code.

For a subset of our queries (the ones sourced from LeetCode and humanEval), we have associated method signatures for the expected result; for LeetCode

sourced queries, we have both Python and Java, and for humanEval, Python only. Using these method signatures, we generate a new query set of 1,698 queries: 768 Java queries, and 930 Python queries. Again, we repeat the same mining process from Section 5.3 to generate a new set of snippets. However, we modify the Copilot input code; we still provide the natural language query as a comment, but this time provide the method signature in place of the unfinished function starter. For Stack Overflow, we append the method signature to the end of the natural language query.

TABLE 5.10. Overview of the Method Query Dataset

	Queries	**Snippets**		
		SO	**CP**	**Total**
Total	1,698	1,869	9,716	11,585
Java	768	843	4,455	5,298
Python	930	1,026	5,261	6,287

Table 5.10 shows the number of snippets with modified queries. For the same subset of queries, the original dataset returns 10,102 Stack Overflow snippets and 11,519 Copilot snippets, thus we see an 81.50% reduction in the number Stack Overflow snippets, and a 15.65% decrease for Copilot. Snippets per query average 1.10 for Stack Overflow, while for Copilot the average is 5.72 snippets. However, Stack Overflow fails to generate results for 1,443 queries (84.98%). Copilot only fails for 1 query. Again, Copilot removes 7,317 duplicates, across 87.87% of queries. When we look at exact matches between the unmodified and modified per query snippet sets, we find only 393 incidences for the Stack Overflow set and 48 for the Copilot set. For 10.01% of Stack Overflow queries, at least one snippet remains the same. The NCD between snippets of the same query did not change significantly with the addition of method calls.

Finding 9: Method Signatures

Stack Overflow returns 81.50% fewer snippets when queries include method signatures, and fails to generate output for 1,443 of the 1,698 queries. For Copilot, the reduction in snippets was smaller: 15.65%.

These findings suggest, as expected, that traditional web search does not work well with code. The implication of this, then, is that LLMs like Copilot that are capable of utilizing code context enable a new way to search for code snippets. If developers have some idea of what a method input and output should be, providing method signature information can be useful. However, Copilot's lack of diversity remains an issue; 87.87% of queries had duplicates that were removed.

5.4.10 Threats to Validity

There are several potential threats to the validity of this study.

Threats to the *internal validity* compromise our accuracy in establishing relationships between independent and dependent variables. The selection of queries and code snippets used for our evaluation may not be representative of the diverse range of programming problems developers face. We tried to mitigate this threat by choosing programming problems from a range of sources. Additionally, the metrics used to evaluate output quality and user interface, such as NCD, might not fully capture the nuances of code quality or relevance. To reduce the impact of this threat, we compared multiple aspects of the code snippets. Additionally, we cannot be sure to what extent Stack Overflow threads were part of the training data for Copilot, and we do not know how that might have affected our results.

Threats to *external validity* relate to the generalizability of our results. We cannot generalize our results to other languages, ecosystems, or to other websites or large language models. The study was also conducted with specific versions of Copilot and a particular snapshot of Stack Overflow. Updates over time may lead to different results, limiting the long-term validity of the findings. Future iterations of large language models could address some of the weaknesses identified in the study, rendering some conclusions obsolete.

Despite these threats, our findings provide insights into the ongoing shift from copy-paste programming to a growing reliance on AI-generated code, highlighting the utility and limitations of generated code in comparison to conventional methods.

5.5 Conclusion

We systematically examined the strengths and weaknesses of alternative code completion approaches, Copilot and Stack Overflow, in the context of software development. Our findings reveal that Copilot generates snippets more frequently, has fewer syntax errors, and provides more helpful code compared to Stack Overflow. However, Copilot's output is less diverse, tends to hallucinate answers, and has fewer API calls. Stack Overflow snippets, on the other hand, contain more comments and are more diverse per query. Query variations, such as using keywords or method signatures, have different effects on both platforms. These insights shed light on the trade-offs between Copilot and Stack Overflow, providing guidance for software professionals seeking to navigate the rapidly evolving landscape of source code reuse. Understanding these trade-offs will facilitate more informed decision-making and contribute to the development of efficient and intelligent programming practices.

Chapter 6

Conclusion and Future Work

The automation of code reuse is a promising direction of research in the field of software engineering, where much development relies on the reuse of already existing software artefacts, such as code snippets and libraries. Developers source code snippets from various sources, such as online or those generated by large language models, and each presents unique challenges. The overall aim of this work is to improve the efficiency of software developers by automating existing workflows; dedicated tools can improve efficiency, for example, by automatically correcting errors in the context of the intended use of the snippet. In this thesis, we investigate various techniques for automating aspects of the code reuse process and evaluate these techniques — in one case, with real developers.

Existing work in this area often focuses on automating one part of the code reuse process; that is typically the retrieval of code snippets. As we describe in Chapter 1, searching is only the first stage of the code reuse process developers usually undertake. This still leaves developers grappling with code that often does not run, and that may require correction and integration within their existing project. We propose a solution, Node Code Query (NCQ), that both embeds code and library search within the programming environment *and* automatically corrects errors in snippets. NCQ improves code snippet quality and aids developers in the integration of snippets. Following this work, we investigate a technique for more robust code snippet error detection and improvement in Node.js using the TypeScript compiler, which out-performs our previous parser-based approach. Finally, we examine how AI code synthesis such as OpenAI's Codex and GitHub's Copilot fit into the code reuse discussion, when generated code aims to fulfil a similar purpose as online code snippet reuse. We compare Copilot to current reuse patterns that utilize Google and Stack Overflow and find that Copilot generates snippets with fewer errors, likely because it is capable of taking the code context into account when generating. In this way, large language model code generation may reduce the need for developers to correct and integrate snippets, making them useful for developers already despite growing concerns for the quality of their output.

This thesis makes the following contributions:

- The development of a Node.js read-evaluate-print loop (REPL) programming environment, Node Code Query (NCQ), that embeds library and code search within the editor to reduces context switching between editor and web browser. Additionally, NCQ automatically corrects snippets

before showing them to developers, to reduce time spent manually correcting them. We evaluate NCQ and its prototype with 20 Node.js developers against baselines of manual code reuse. Our results show that our code reuse focused programming environment and snippet correction aids developers in finding solutions to their tasks more quickly than manual code reuse, and that developer perceptions of the tool are generally positive.

- Based on the results of NCQ and an analysis of existing work in Node.js error detection and correction, we investigate the use of the TypeScript compiler for the automated correction of code snippets. We observe that existing work in this area relies on parsers and linters, which often fail in cases of erroneous code. When we observe the results of ESLint on our dataset of NPM README code snippets, we find that many snippets report only a single parsing error (indicating where parsing failed), and for these cases, no AST can be generated. We improve upon NCQ's code corrections to produce Node Code Correction (NCC), our approach for correcting Node.js code snippets using the TypeScript compiler. With the TS Compiler, we observe that we can detect a greater range of errors, with more useful information. With the use of TypeScript's built-in fixes, our own custom fixes and line deletion, we are able to reduce the number of snippets with errors from 73.7% to 25.1%.

- We compare large language model (LLM) code generation to existing manual code snippet reuse processes, via a comparison of Copilot and Stack Overflow combined with Google. LLM generated code threatens to change the landscape of both code reuse and software development. In particular, generated snippets serve a similar purpose as online-sourced snippets; when a developer wants to find a code solution to a task, they may no longer have to look online, but instead generate them for their unique context. However, questions remain about the quality and usefulness of its output at this stage. In this sense, we aim to characterise the differences between Copilot's output and Stack Overflow's output for the same set of queries, and what this might tell us about their usefulness going forward. We observe that Copilot snippets had fewer errors, but less diversity, and they were not without problems that might still slow the adoption of LLM code completion.

The work in this thesis represents one step towards automating code reuse, but there are multiple directions for future work. In general, there is room to improve the results of our approaches further. Additionally, like many of the techniques we present build on existing work and adapt techniques for Node.js (such as our prior work in Java), there may again be unique challenges adapting them for other languages. Because code search and error detection and correction are often language specific — that is, the characteristics of different languages make error correction non-generalisable, and developers in different language ecosystems have their own sources for snippets and libraries — there is always the potential to adapt these techniques for new contexts and languages. The following sections describe other avenues for future work.

6.1 Empirical Studies on Code Correction

There is currently little understanding of the changes developers make when manually correcting snippets during the code reuse process. Of interest is how useful automated corrections like the ones employed by NCQ and NCC are to real developers. In Chapter 4 we evaluate NCC on a dataset of real Stack Overflow snippets over time, to compare to manual code corrections, and in Chapter 3 we evaluate code corrections within the context of our tool. However, we have yet to investigate how developers actually fix snippets, and how comparable our automated approaches are. By studying how real developers correct snippets, and what they think of our automated fixes, we might uncover new insights to improve our approaches.

One potential study design may be to source a dataset of erroneous snippets (such as the dataset used to evaluate NCC in Chapter 4) and have developers correct these snippets in a study, observing the steps they take. From this study we would additionally produce a dataset of manually corrected snippets that we could then compare to our automatic fixes via NCC. Additionally, to observe what developers think of our automatic fixes, we could run a second study with another set of developers, where we ask them to provide feedback for snippets from both sources. The aim would be to see if feedback differs between the two sources, and why. Alternatively, we could observe developers using NCC as part of the code reuse process, and compare to manual reuse, in order to investigate how useful the fixes are to developers within this context. We would look at time to complete various programming tasks with and without assistance, how often developers accept our automatic corrections and how often they need to make additional changes.

6.2 Empirical Study of GitHub Copilot

In Chapter 5 we compare the output of Copilot to Stack Overflow combined with Google, in order to gain insights into their differences and how large language model code generation may be used in the future. However, we may gain additional insights by comparing their use with actual developers. Ultimately, it is developer perceptions of these tools that will shape their future adoption. By observing developers using the plug-in in real programming situations, and by asking for their perspectives, we may identify new concerns that other research has overlooked. Additionally, we may be interested in the *awareness* of existing concerns in the literature, and what developers do to mitigate what problems they currently experience. This knowledge can help drive future development of similar tools.

Already, there is a large amount of research on prompt engineering, or how users interface with large language models to get their desired output. However, there has been little empirical work on GitHub Copilot, which might reveal if this research holds true in a software development context, and in actual use. In Chapter 5 we observe, for example, that our prompt location impacted the types of snippets we could generate; no Java Copilot snippet included an import

statement (because import statements need to be at the top of the file). By observing real use, we may reveal more cases like this, or observe how frequent situations like this actually are. While we currently conjecture that Copilot's snippets fulfil a similar purpose as online code reuse, it's possible that the situations that developers employ both techniques might also differ.

A proposed study idea is to have real developers complete programming tasks with the help of either Copilot and Stack Overflow, and then compare their performance. Additionally, we would record and log their development process, to see how often they searched for snippets with either resource, how often they accept the first result, and how often they needed to make changes to a given snippet (and the nature of those changes), as well as the types of snippets they seek out (fragments vs whole functions or classes) and the types of queries they use. We would aim to select a variety of tasks that cover various concerns with LLM generated code in the literature, such as security and library usage. While we hypothesise that Copilot and Stack Overflow are used in similar situations, the additional convenience and auto-complete aspects of Copilot may make usage more frequent; on the other hand, quality concerns may make developers more hesitant.

Additionally, we would be interested in developer perceptions of both approaches. Developers might already use or avoid Copilot, or prefer one approach over the other, and the reasons they give can aid development of future tools. For example, if developers seem to generally like that Copilot snippets are generated for them as they program, this is potentially a feature that could be added to a code reuse tool. Conversely, if developers prefer Stack Overflow snippets because they're accompanied by natural language explanations and feedback, such as votes and comments that help them make informed decisions, perhaps there is an avenue for combining generated code with similar aspects. It would also be interesting to observe the current usage rates of Copilot in comparison to Stack Overflow.

6.3 AI Assisted Code Correction and Integration of Snippets

One potential avenue of merging the benefits of online code reuse and large language models may be to use LLMs to perform the integration of existing snippets. In Chapter 4 we note that much work in the area of correcting snippets for new contexts relies on heuristic fixes that must make guesses; for example, when defining a missing variable, it must have its value guessed. A LLM has the ability to predict what this value should be based on context and from the large amount of code in its training data. For these reasons, using LLMs to integrate and correct snippets during the reuse process may be a good fit. Additionally, we observed in Chapter 5 that snippets generated by Copilot had fewer errors than those sourced from Stack Overflow, but that they lacked diversity and could be nonsensical at times. One issue identified with LLMs is the generation of nonsense output, or 'hallucinations', so we ask if output is more likely to be 'grounded' if the model operates on an existing code input. Sourcing snippets

from online may avoid some concerns with LLM output quality. This idea aims to leverage both the usefulness of LLM generated snippets and online snippets together.

A potential research idea is to develop a tool that can take a code snippet to insert, and the destination file and location, combine these together and then prompt a LLM (either something code-focused such as Codex or general purpose like GPT) to fix any integration errors. To validate the tool, we could then see how the error landscape changes before and after using static analysis, like the methods discussed in Chapter 4. Additionally, we could observe the kinds of changes that a LLM makes to snippets, and if the general concerns with LLM generated code remain true in this new context. It may also be interesting to see if providing the results of static analysis, such as detected errors in a piece of code, improves performance.

6.4 Automatically Generated Test Cases

Test cases are an important aspect of validating code functionality, especially to determine the runnability and quality of code, but this has often been under-utilised in work relating to code snippets. While most code snippets sourced from online do not run, there is a desire to measure their functionality when they do work.

In prior work on NLP2TestableCode (Reid, Treude, and Wagner, 2020), we attempted to generate JUnit test cases for Java code snippets sourced from Stack Overflow using heuristics. This process involves converting a snippet to a function with input and output, where these values are guessed using heuristics. Then, potential output values are guessed for a test case. These test cases were used as part of a code reuse tool in an attempt to automate the functionality testing stage of code reuse, however, this process leaves much to be desired.

As with heuristic corrections, heuristic test case generation could be augmented with LLMs. We would devise a tool to take a snippet, and then generate a function and test case. This tool could also be useful outside a code reuse context, in generating test cases for a given file automatically. In a code reuse context, we would also likely have a natural language query that could be used to derive what the test case should be testing for, which could also be provided. To validate this tool, we may want to test it with real developers to see how useful it is. We may also want to manually evaluate individual outputs to see if they make sense; that is, do they work, and would they be helpful? Finally, we could evaluate the test case generation of a dataset of snippets to see how many are runnable, and how many snippets pass. If we were to develop a tool for Java, we could potentially compare results with our previous approach, where we would expect better performance with the aid of LLMs.

6.5 Investigating Industry Perceptions of Open Source Software Libraries

As discussed in Chapter 1, much of modern software relies on open-source libraries and complicated chains of dependencies, including software in industry (for example, companies such as Microsoft and Meta, who even open-source some of their software). The use of software libraries, like code reuse in general, enables focus elsewhere and speeds up development. However, this reliance on external software has its own challenges. Additionally, there is little work on what industry developers using open-source think about the libraries they frequently use, and what policies they follow and challenges they may experience. Understanding these processes and challenges will enable us to better aid developers when they use libraries in an industry context.

One potential research idea is an interview study of industry developers who use open-source libraries, investigating the processes, policies and challenges of using open-source libraries. We would focus on the areas of adopting libraries, maintenance, security, migration, updating, bugs and licences. We are also interested in what tools developers currently use, or are aware of, to find libraries and track vulnerabilities and updates. For example, in the area of library adoption, we are interested in what qualities developers look at when selecting libraries. We would ask developers about policies and processes, but also present them with situations they may have encountered to understand how they handle similar challenges. One idea is to focus on Node.js and Java developers, as there is a large open source community in both of these ecosystems. Potentially, we could look to contrast the two groups.

The aim of this work would be to identify challenges that could then be addressed in future work. Specifically, we are looking for developers to identify issues they need solutions to that are currently unidentified in the literature. For example, if developers have trouble finding suitable libraries for industry use, there is the potential to develop a tool specifically for finding libraries in this context.

In conclusion, code reuse is an ever-evolving field, and thus the problem of aiding developers in this common development is likely to always require more investigation. As new technologies emerge, so too should complementary tools and techniques to aid reuse. It is also important to understand how developers actually interact with code, libraries and reuse tools, in order to deliver realistic solutions. Now, with the emergence of large language model code generation, the area of code reuse is poised to shift; we hope to give developers the insights and tools to make the most of it.

Bibliography

Abdalkareem, Rabe, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab (2017). "Why Do Developers Use Trivial Packages? An Empirical Case Study on NPM". In: *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 385–395.

Al Madi, Naser (2022). "How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot". In: *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–5.

Alkaissi, Hussam and Samy I McFarlane (2023). "Artificial hallucinations in ChatGPT: implications in scientific writing". In: *Cureus* 15.2.

An, Le, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol (2017). "Stack overflow: A code laundering platform?" In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 283–293.

Asare, Owura, Meiyappan Nagappan, and N Asokan (2022). "Is github's copilot as bad as humans at introducing vulnerabilities in code?" In: *arXiv preprint arXiv:2204.04741*.

Baltes, Sebastian and Stephan Diehl (2019). "Usage and attribution of Stack Overflow code snippets in GitHub projects". In: *Empirical Software Engineering (EMSE)* 24.3, pp. 1259–1295. ISSN: 1573-7616. DOI: 10.1007/s10664-018-9650-5. URL: https://doi.org/10.1007/s10664-018-9650-5.

Baltes, Sebastian, Lorik Dumani, Christoph Treude, and Stephan Diehl (2018). "SOTorrent: reconstructing and analyzing the evolution of stack overflow posts". In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. Association for Computing Machinery, pp. 319–330. DOI: 10.1145/3196398.3196430. URL: https://doi.org/10.1145/3196398.3196430.

Baltes, Sebastian, Richard Kiefer, and Stephan Diehl (2017). "Attribution required: Stack overflow code snippets in GitHub projects". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, pp. 161–163.

Baltes, Sebastian and Markus Wagner (2020). "An Annotated Dataset of Stack Overflow Post Edits". In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. GECCO '20. Cancún, Mexico: Association for Computing Machinery, 1923–1925. ISBN: 9781450371278. DOI: 10.1145/3377929.3398108. URL: https://doi.org/10.1145/3377929.3398108.

Basili, Victor R, Lionel C Briand, and Walcélio L Melo (1996). "How reuse influences productivity in object-oriented systems". In: *Communications of the ACM* 39.10, pp. 104–116.

Begel, Andrew and Thomas Zimmermann (2014). "Analyze This! 145 Questions for Data Scientists in Software Engineering". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 12–23. ISBN: 9781450327565. DOI: 10.1145/2568225.2568233. URL: https://doi.org/10.1145/2568225.2568233.

Bird, Christian, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit (2022). "Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools". In: *Queue* 20.6, pp. 35–57.

Bird, D. L. and C. U. Munoz (1983). "Automatic generation of random self-checking test cases". In: *IBM Systems Journal* 22.3, pp. 229–245. DOI: 10.1147/sj.223.0229.

Brandt, Joel, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer (2010). "Example-centric programming: integrating web search into the development environment". In: *Conference on Human Factors in Computing Systems (CHI)*, pp. 513–522.

Brownlee, Alexander E. I., Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White (2019). "Gin: Genetic Improvement Research Made Easy". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. New York, NY, USA: Association for Computing Machinery, 985–993. ISBN: 9781450361118. DOI: 10.1145/3321707.3321841. URL: https://doi.org/10.1145/3321707.3321841.

Cai, Liang, Haoye Wang, Bowen Xu, Qiao Huang, Xin Xia, David Lo, and Zhenchang Xing (2019). "AnswerBot: an answer summary generation tool based on stack overflow". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1134–1138.

Campbell, Brock Angus and Christoph Treude (2017). "NLP2Code: Code Snippet Content Assist via Natural Language Tasks". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 628–632.

Campos, Uriel Ferreira, Guilherme Smethurst, Joao Pedro Moraes, Rodrigo Bonifácio, and Gustavo Pinto (2019). "Mining rule violations in javascript code snippets". In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. IEEE, pp. 195–199.

Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. (2021). "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374*.

Chinthanet, Bodin, Brittany Reid, Christoph Treude, Markus Wagner, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto (2021). *What makes a good Node.js package? Investigating Users, Contributors, and Runnability*. arXiv: 2106.12239 [cs.SE].

Cilibrasi, Rudi and Paul MB Vitányi (2005). "Clustering by compression". In: *IEEE Transactions on Information theory* 51.4, pp. 1523–1545.

Cockburn, Andy, Carl Gutwin, and Saul Greenberg (2007). "A Predictive Model of Menu Performance". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. New York, NY, USA: Association for Computing Machinery, 627–636. ISBN: 9781595935939. DOI: 10.1145/1240624.1240723. URL: https://doi.org/10.1145/1240624.1240723.

Cottrell, Rylan, Robert J. Walker, and Jörg Denzinger (2008). "Jigsaw: A Tool for the Small-Scale Reuse of Source Code". In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion '08. New York, NY, USA: Association for Computing Machinery, 933–934. ISBN: 9781605580791. DOI: 10.1145/1370175.1370194. URL: https://doi.org/10.1145/1370175.1370194.

Dakhel, Arghavan Moradi, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, Zhen Ming, et al. (2022). "GitHub Copilot AI pair programmer: Asset or Liability?" In: *arXiv preprint arXiv:2206.15331*.

Danilova, Anastasia, Alena Naiakshina, Stefan Horstmann, and Matthew Smith (2021). "Do you Really Code? Designing and Evaluating Screening Questions for Online Surveys with Programmers". In: *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 537–548. DOI: 10.1109/ICSE43902.2021.00057.

Davis, Fred D (1989). "Perceived usefulness, perceived ease of use, and user acceptance of information technology". In: *Management Information Systems (MIS) Quarterly*, pp. 319–340.

Duran, Joe W. and Simeon C. Ntafos (1984). "An Evaluation of Random Testing". In: *IEEE Transactions on Software Engineering* SE-10.4, pp. 438–444. DOI: 10.1109/TSE.1984.5010257.

El-Hajj, Rehab and Sarah Nadi (2020). "LibComp: An IntelliJ Plugin for Comparing Java Libraries". In: *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Virtual Event, USA, 1591–1595. ISBN: 9781450370431.

Erhabor, Daniel (2022). "Measuring the Performance of Code Produced with GitHub Copilot". MA thesis. University of Waterloo.

ESLint (2023). *Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter*. URL: https://eslint.org/.

Fraser, Gordon and Andrea Arcuri (2011). "Evosuite: automatic test suite generation for object-oriented software". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416–419.

Fry, Zachary P, Bryan Landau, and Westley Weimer (2012). "A human study of patch maintainability". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 177–187.

Ginelli, Davide, Matias Martinez, Leonardo Mariani, and Martin Monperrus (2022). "A comprehensive study of code-removal patches in automated program repair". In: *Empirical Software Engineering* 27.4, p. 97. DOI: 10.1007/s10664-021-10100-7. URL: https://doi.org/10.1007/s10664-021-10100-7.

GitHub (2023). *GitHub Copilot - Your AI pair programmer*. URL: https://github.com/features/copilot.

Gu, Xiaodong, Hongyu Zhang, and Sunghun Kim (2018). "Deep code search". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, pp. 933–944.

Hanford, Kenneth V. (1970). "Automatic generation of test cases". In: *IBM Systems Journal* 9.4, pp. 242–257.

Harrand, Nicolas, Simon Allier, Marcelino Rodriguez-Cancio, Martin Monperrus, and Benoit Baudry (2019). "A journey among Java neutral program variants". In: *Genetic Programming and Evolvable Machines* 20.4, pp. 531–580. DOI: 10.1007/s10710-019-09355-3. URL: https://doi.org/10.1007/s10710-019-09355-3.

Heinemann, Lars, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck (2011). "On the extent and nature of software reuse in open source java projects". In: *International Conference on Software Reuse*. Springer, pp. 207–222.

Horton, Eric and Chris Parnin (2018). "Gistable: Evaluating the executability of python code snippets on github". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 217–227.

Imai, Saki (2022). "Is GitHub copilot a substitute for human pair-programming? An empirical study". In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 319–321.

Jansen, Bernard J, Amanda Spink, and Tefko Saracevic (2000). "Real life, real users, and real needs: a study and analysis of user queries on the web". In: *Information processing & management* 36.2, pp. 207–227.

Jaworski, Mateusz and Dariusz Piotrkowski (2023). "Study of software developers' experience using the Github Copilot Tool in the software development process". In: *arXiv preprint arXiv:2301.04991*.

Ji, Ziwei, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung (2023). "Survey of hallucination in natural language generation". In: *ACM Computing Surveys* 55.12, pp. 1–38.

Ji, Ziwei, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung (2022). "Survey of hallucination in natural language generation". In: *ACM Computing Surveys*.

Jupyter (2020). URL: https://jupyter.org/.

Kip (2009). *Best way to represent a fraction in Java? — stackoverflow.com*. https://stackoverflow.com/a/474612/13988074. [Accessed 30-Mar-2023].

Kitchenham, Barbara and Shari Pfleeger (Jan. 2008). "Personal Opinion Surveys". In: pp. 63–92. ISBN: 978-1-84800-043-8. DOI: 10.1007/978-1-84800-044-5_3.

Kula, Raula Gaikovina, Ali Ouni, Daniel M German, and Katsuro Inoue (2017). "On the impact of micro-packages: An empirical study of the npm javascript ecosystem". In: *arXiv:1709.04638*.

Kula, Raula Gaikovina and Christoph Treude (2022). "In war and peace: the impact of world politics on software ecosystems". In: *Proceedings of the 30th*

ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1600–1604.

Lajkó, Márk, Viktor Csuvik, and László Vidács (2022). "Towards JavaScript program repair with Generative Pre-trained Transformer (GPT-2)". In: *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pp. 61–68. DOI: 10.1145/3524459.3527350. URL: https://doi.org/10.1145/3524459.3527350.

Larios Vargas, Enrique, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios (2020). "Selecting third-party libraries: The practitioners' perspective". In: *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 245–256.

Lawrence, Lizzy (2022). *'You're reading my mind': GitHub's AI coder amazes (and terrifies) developers*. https://www.protocol.com/workplace/github-copilot-ai-developers.

Lazzarini Lemos, Otávio Augusto, Sushil Krishna Bajracharya, and Joel Ossher (2007). "Codegenie: a tool for test-driven source code search". In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 917–918.

Le Goues, Claire, Westley Weimer, and Stephanie Forrest (2012). "Representations and Operators for Improving Evolutionary Software Repair". In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. GECCO '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 959–966. ISBN: 9781450311779. DOI: 10.1145/2330163.2330296. URL: https://doi.org/10.1145/2330163.2330296.

Licorish, Sherlock A. and Markus Wagner (2022a). "Combining GIN and PMD for Code Improvements". In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. Boston, Massachusetts, 790–793.

Licorish, Sherlock A. and Markus Wagner (2022b). "Dissecting Copy/Delete/Replace/Swap Mutations: Insights from a GIN Case Study". In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. GECCO '22. Boston, Massachusetts: Association for Computing Machinery, 1940–1945. ISBN: 9781450392686. DOI: 10.1145/3520304.3533970. URL: https://doi.org/10.1145/3520304.3533970.

Lim, W. C. (Sept. 1994). "Effects of reuse on quality, productivity, and economics". In: *IEEE Software* 11.5, pp. 23–30. ISSN: 0740-7459. DOI: 10.1109/52.311048.

Lipow, Myron (1982). "Number of faults per line of code". In: *IEEE Transactions on software Engineering* 4, pp. 437–439.

Liu, Jiakun, Sebastian Baltes, Christoph Treude, David Lo, Yun Zhang, and Xin Xia (2021). "Characterizing search activities on stack overflow". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 919–931.

Lo, David, Nachiappan Nagappan, and Thomas Zimmermann (2015). "How Practitioners Perceive the Relevance of Software Engineering Research". In:

Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 415–425. ISBN: 9781450336758. DOI: 10.1145/2786805.2786809. URL: https://doi.org/10.1145/2786805.2786809.

Mahajan, Sonal, Negarsadat Abolhassani, and Mukul Prasad (Nov. 2020). "Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching". In: pp. 1052–1064. DOI: 10.1145/3368089.3409764.

Makyen (2023). *Temporary policy: ChatGPT is banned — meta.stackoverflow.com*. https://meta.stackoverflow.com/questions/421831/temporary-policy-chatgpt-is-banned. [Accessed 30-Mar-2023].

Mastropaolo, Antonio, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota (2023). "On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot". In: *arXiv preprint arXiv:2302.00438*.

McIlroy, M Douglas, J Buxton, Peter Naur, and Brian Randell (1968). "Mass-produced software components". In: *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*, pp. 88–98.

Meldrum, Sarah, Sherlock A Licorish, Caitlin A Owen, and Bastin Tony Roy Savarimuthu (2020). "Understanding stack overflow code quality: A recommendation of caution". In: *Science of Computer Programming* 199, p. 102516.

Melo, L., I. S. Wiese, and M. d'Amorim (2019). "Using Docker to Assist Q A Forum Users". In: *IEEE Transactions on Software Engineering (TSE)*.

Microsoft (2023a). *ts-fix*. URL: https://github.com/microsoft/ts-fix.

Microsoft (2023b). *TypeScript: JavaScript With Syntax For Types*. URL: https://www.typescriptlang.org/.

Microsoft (2023c). *Working with JavaScript in Visual Studio Code*. URL: https://code.visualstudio.com/docs/nodejs/working-with-javascript.

Mirhosseini, Samim and Chris Parnin (2020). "Docable: Evaluating the Executability of Software Tutorials". In: *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Virtual Event, USA, 375–385.

Mockus, Audris (2007). "Large-scale code reuse in open source software". In: *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, pp. 7–7.

Mohagheghi, Parastoo, Reidar Conradi, Ole M Killi, and Henrik Schwarz (2004). "An empirical study of software reuse vs. defect-density and stability". In: *Proceedings. 26th International Conference on Software Engineering*. IEEE, pp. 282–291.

Morgan, Rachael (2015). "Analysing and comparing problem landscapes for black-box optimization via length scale". PhD thesis.

Nguyen, Nhan and Sarah Nadi (2022). "An empirical evaluation of GitHub copilot's code suggestions". In: *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 1–5.

Ocariza Jr, Frolin S, Karthik Pattabiraman, and Ali Mesbah (2014). "Vejovis: Suggesting fixes for JavaScript faults". In: *Proceedings of the 36th International Conference on Software Engineering*, pp. 837–847.

OpenAI, Wojciech Zaremba, and Greg Brockman (2021). *OpenAI Codex*. URL: https://openai.com/blog/openai-codex.

Pacheco, Carlos, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball (2007). "Feedback-directed random test generation". In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE, pp. 75–84.

Pearce, H., B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri (May 2022). "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions". In: *2022 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 980–994. DOI: 10.1109/SP46214.2022.00057. URL: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00057.

Peng, Sida, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer (2023). "The impact of ai on developer productivity: Evidence from github copilot". In: *arXiv preprint arXiv:2302.06590*.

Petke, Justyna, Brad Alexander, Earl T. Barr, Alexander E. I. Brownlee, Markus Wagner, and David R. White (2019). "A Survey of Genetic Improvement Search Spaces". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. Prague, Czech Republic: Association for Computing Machinery, 1715–1721. ISBN: 9781450367486. DOI: 10.1145/3319619.3326870. URL: https://doi.org/10.1145/3319619.3326870.

Pham, Tri Minh Triet and Jinqiu Yang (2020). "The secret life of commented-out source code". In: *Proceedings of the 28th International Conference on Program Comprehension*, pp. 308–318.

PMD (2023). *PMD*. URL: https://pmd.github.io/.

Ponzanelli, Luca, Alberto Bacchelli, and Michele Lanza (2013). "Seahawk: Stack overflow in the ide". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, pp. 1295–1298.

Ponzanelli, Luca, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza (2014). "Prompter: A self-confident recommender system". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 577–580.

Project, NLTK (2023). *NLTK :: Natural Language Toolkit — nltk.org*. https://www.nltk.org/. [Accessed 30-Mar-2023].

Proksch, Sebastian, Veronika Bauer, and Gail C. Murphy (2015). "How to Build a Recommendation System for Software Engineering". In: *Software Engineering: International Summer Schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Springer, pp. 1–42. ISBN: 978-3-319-28406-4.

Puryear, Ben and Gina Sprint (2022). "Github copilot in the classroom: learning to code with AI assistance". In: *Journal of Computing Sciences in Colleges* 38.1, pp. 37–47.

Ragkhitwetsagul, Chaiyong, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto (2021). "Toxic Code Snippets on Stack Overflow". In: *IEEE Transactions on Software Engineering* 47.3, pp. 560–581. DOI: 10.1109/TSE.2019.2900307.

Rainer, Austen and Claes Wohlin (2021). *Recruiting credible participants for field studies in software engineering research*. DOI: 10.48550/ARXIV.2112.14186. URL: https://arxiv.org/abs/2112.14186.

Reid, Brittany, Earl Barr, Markus Wagner, and Christoph Treude (2023a). *Copy-Paste vs. Code Generation: Contrasting Stack Overflow and GitHub Copilot*. Being revised for submission to IEEE Transactions on Software Engineering.

Reid, Brittany, Marcelo d'Amorim, Markus Wagner, and Christoph Treude (2023b). "NCQ: Code Reuse Support for Node.js Developers". In: *IEEE Transactions on Software Engineering* 49.5, pp. 3205–3225. DOI: 10.1109/TSE.2023.3248113.

Reid, Brittany, Christoph Treude, and Markus Wagner (2020). "Optimising the Fit of Stack Overflow Code Snippets into Existing Code". In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 1945–1953.

Reid, Brittany, Christoph Treude, and Markus Wagner (2023). "Using the TypeScript compiler to fix erroneous Node.js snippets". To appear in: *23rd IEEE International Working Conference on Source Code Analysis and Manipulation*.

Reid, Brittany, Markus Wagner, Marcelo d'Amorim, and Christoph Treude (2022). *Software Engineering User Study Recruitment on Prolific: An Experience Report*. DOI: 10.48550/ARXIV.2201.05348. URL: https://arxiv.org/abs/2201.05348.

Sandewall, Erik (1978). "Programming in an Interactive Environment: the "Lisp" Experience". In: *ACM Computing Surveys (CSUR)* 10.1, pp. 35–71.

Sobania, Dominik, Martin Briesch, and Franz Rothlauf (2022). "Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming". In: *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1019–1027.

Sojer, Manuel and Joachim Henkel (2010). "Code reuse in open source software development: Quantitative evidence, drivers, and impediments". In: *Journal of the Association for Information Systems* 11.12, pp. 868–901.

Stack Exchange (2023). *Stack Exchange Data Explorer*. URL: https://data.stackexchange.com/.

Steinmacher, Igor, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa (2016). "Overcoming open source project entry barriers with a portal for newcomers". In: *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 273–284.

Storey, Margaret-Anne, Christoph Treude, Arie van Deursen, and Li-Te Cheng (2010). "The impact of social media on software engineering practices and tools". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 359–364.

Tal, Liran and Simon Maple (2020). *npm passes the 1 millionth package milestone! What can we learn?* URL: https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn/.

Terragni, Valerio, Yepang Liu, and Shing-Chi Cheung (2016). "CSNIPPEX: automated synthesis of compilable code snippets from Q&A sites". In: *Proceedings of the 25th international symposium on software testing and analysis*, pp. 118–129.

Tonella, Paolo (2004). "Evolutionary testing of classes". In: *ACM SIGSOFT Software Engineering Notes* 29.4, pp. 119–128.

Tufano, Michele, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan (2022). "Generating accurate assert statements for unit test cases using pretrained transformers". In: *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pp. 54–64.

Umarji, Medha, Susan Elliott Sim, and Crista Lopes (2008). "Archetypal internet-scale source code searching". In: *IFIP International Conference on Open Source Systems*. Springer, pp. 257–263.

Vaithilingam, Priyan, Tianyi Zhang, and Elena L Glassman (2022). "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models". In: *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7.

Venkatesh, Viswanath and Fred D Davis (2000). "A theoretical extension of the technology acceptance model: Four longitudinal field studies". In: *Management Science* 46.2, pp. 186–204.

W3Techs (2020). *Usage Statistics of Node.js*. URL: https://w3techs.com/technologies/details/ws-nodejs.

Wang, Junjie, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang (2023). "Software Testing with Large Language Model: Survey, Landscape, and Vision". In: *arXiv preprint arXiv:2307.07221*.

Wong, Dakota, Austin Kothig, and Patrick Lam (2022). "Exploring the Verifiability of Code Generated by GitHub Copilot". In: *arXiv preprint arXiv:2209.01766*.

Xu, Frank F., Bogdan Vasilescu, and Graham Neubig (Mar. 2022). "In-IDE Code Generation from Natural Language: Promise and Challenges". In: *Transactions on Software Engineering and Methodology (TOSEM)* 31.2. ISSN: 1049-331X. DOI: 10.1145/3487569. URL: https://doi.org/10.1145/3487569.

Yang, Di, Aftab Hussain, and Cristina Videira Lopes (2016). "From Query to Usable Code: An Analysis of Stack Overflow Code Snippets". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. New York, NY, USA: Association for Computing Machinery, 391–402. ISBN: 9781450341868. DOI: 10.1145/2901739.2901767. URL: https://doi.org/10.1145/2901739.2901767.

Yetistiren, Burak, Isik Ozsoy, and Eray Tuzun (2022). "Assessing the quality of GitHub copilot's code generation". In: *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 62–71.

Yu, Sehun, Dongha Lee, and Hwanjo Yu (2021). "Convolutional neural networks with compression complexity pooling for out-of-distribution image detection". In: *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pp. 2435–2441.

Zhang, Beiqi, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem (2023). "Practices and Challenges of Using GitHub Copilot: An Empirical Study". In: *arXiv preprint arXiv:2303.08733*.

Zhang, Hongyu, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu (2016). "Bing developer assistant: improving developer productivity by recommending sample code". In: *International Symposium on the Foundations of Software Engineering (FSE)*, pp. 956–961.

Ziegler, Albert, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian (2022). "Productivity assessment of neural code completion". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 21–29.