Optimizing Asynchronous JavaScript Applications

by

Alexi Turcotte

A thesis
presented to Northeastern University
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Boston, Massachusetts, USA, 2023

© Alexi Turcotte 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiners:	Ali Mesbah Professor, University of British Columbia Andreas Zeller Professor, CISPA Helmholtz Centre for Information Security
Supervisors:	Frank Tip Professor, Khoury College, Northeastern University Jan Vitek Professor, Khoury College, Northeastern University
Internal Members:	Jonathan Bell Asst. Professor, Khoury College, Northeastern University Arjun Guha Assoc. Professor, Khoury College, Northeastern University

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

JavaScript is arguably today's most popular programming language, and it is ubiquitous as the "language of the web". It is dynamically typed, meaning that programmers do not write type annotations, and beyond this it also has a nonrestrictive dynamic semantics. This makes it easy for programmers to write code that runs, though determining if the code is correct or efficient is an entirely different story. Concretely, JavaScript's dynamism renders *sound and precise static analysis of the language extremely difficult.* This complicates the development of tooling for JavaScript which could help programmers write correct and efficient code.

Sound and precise analysis of JavaScript is beyond the state of the art, and in this thesis we explore the effectiveness of using *unsound* analysis to build tools to detect and remediate inefficiencies in asynchronous JavaScript programs. We explore the following thesis statement: *Unsound analysis of asynchronous JavaScript applications yields actionable insights and effective optimizations.* We support this statement with four approaches to detect and remediate sub-optimal *anti-patterns* in various application domains. Promising results in all cases suggest that perfect is the enemy of good, and that unsound approaches are viable and useful for improving JavaScript code.

Acknowledgements

J'aimerais remercier le département de mathématiques de l'Université Laurentienne. Reposez en paix.

Frank: thank you for your support and encouragement, for being an exemplary mentor, and for your guidance in navigating the research community. Looking forward to many more years of working together.

Jan: thanks for sharing your enthusiasm for cool research and sound science, and for knowing what questions to ask and teaching me to ask the right questions. It was (and will continue to be!) a pleasure working with you.

Committee: thanks to Jon for your enthusiastic support, Arjun for your zany energy, and Ali and Andreas for the interesting discussion (and getting up super early / working late to attend my proposal and defence!).

Amber, Alexander, Phraea, Craig: thanks for all the late night dungeon delving. Remember, every adventure have skeleton.

Aviral, Filip, Ming-Ho, Ben, Julia, Artem, Aaron, Farideh, Satya, Michelle, Max, Mark, Mike, Syndey, Dan, Sam, YT, and many more: thanks for making the lab such a fun place!

All my family: an emphatic thank you for your endless encouragement, for not repurposing my bedroom back home, and for your curiosity and interest in everything I've gotten up to these past years.

Ellen: There's no sense in trying to describe this in words.

This work was supported by NSERC, by Office of Naval Research (ONR) grants N00014-17-1-2945 and N00014-21-1-2491, and by National Science Foundation grants CCF-1715153, CCF-1930604, and CCF-190772.

Dedication

To grandma, your light shines on forever.

To mom, now you can rest.

Table of Contents

Chapter 1

Introduction

JavaScript is arguably today's most popular programming language; according to the "States of the Octoverse" 2022 [98], JavaScript is the most popular language in Github repositories. It is ubiquitous as the *language of the web*, and is used for building client-side scripts, and also back-end servers thanks to the Node.js browserless JavaScript runtime. Modern JavaScript is performant, expressive, and has an incredibly rich ecosystem of packages (`npm`) that support or automate many development tasks.

JavaScript is a dynamically typed programming language, which means that programmers do not write type annotations, but beyond this it is also highly dynamic in its semantics. One would think that the lack of a static type system would result in many more runtime errors, but JavaScript rarely raises such errors due to aggressive coercion of values, and a lax semantics when it comes to object properties. Concretely, basic arithmetic operators are defined on unusual combinations of inputs (e.g., you can add an object and a number, you can concatenate strings with numbers using addition, etc.), accessing properties that are not present on an object yields a value (`undefined`) rather than raising an error, and objects can be extended with new properties at runtime. This makes it easy for programmers to write code that runs, though determining if the code is correct or efficient is an entirely different story.

There is a cost to the dynamism pervading JavaScript: namely, *sound and precise static analysis of JavaScript is extremely difficult*. When an analysis needs to account for all of the potential behavior of JavaScript code, the realm of possibilities quickly becomes so large as to outpace modern processing power. This is unfortunate, as these kinds of analysis could be leveraged to build tools to improve the quality of the immense quantity of JavaScript code running every day.

1.1 Soundness of Program Analysis

Traditionally, a static analysis is said to be *sound* if it reports no *false negatives*; i.e., the analysis does not miss anything. For example: a call graph built using sound analysis will contain all valid call targets for each call site, but may contain superfluous targets; or a sound bug finding tool will detect all bugs, but may also report superfluous bugs by misidentifying correct code as buggy. Soundness is desirable, but plenty of sound analyses are not particularly useful: for example, an analysis that reports that all variables in a program have type "top" or "any" is sound but useless. Unlike sound analysis, *unsound* analysis is free to report false negatives, and of course both can report false positives.

A user's tolerance for false negatives or false positives is highly dependent on the application domain. In security, for example, an analysis that misses no security vulnerabilities is highly desirable; that said, if the analysis reports too many false positives will not be met with enthusiasm as confirming the presence of a security vulnerability in code can be extremely time consuming. In contrast, a linting tool that misses some "code smells" is fine, as the consequences of missing one are far from catastrophic. At a high level, if false negatives can cause serious issues, sound analysis may be best, but developers need to balance false positives with false negatives. Previous work by Sadowski et al. [189, 188] report that a bug finding tool is deemed useful by developers if 90% of the bugs reported by the analysis are indeed bugs; i.e., users are tolerant of a few false positives.

Analysis developers can tinker with the *precision* of an analysis, i.e., how precise is the information reported by the analysis. Imagine a program with the single variable assignment `let v = 5`, and an analysis to determine the type of `v`. The analysis could conclude that `v` has type "top", `number`, `integer`, or even `5` depending on the expressiveness of the type system. These are all *true* statements, but saying that `v` has type `integer` is more *precise* than saying it has type "top". In the realm of building call graphs, a more precise analysis would report fewer potential call targets for a call site than a less precise analysis.

Finding the right level of precision for an analysis is far from straightforward, as it may simply be too costly for an analysis to be both sound *and precise*; this is known as *scalability*. (Note that unsound analyses are free to produce false positives, so they can be made arbitrarily precise by reporting precise but false information.) Given infinite time to exhaust all possible options, a sound analysis can be made precise, but programmers do not have infinite patience, and scalability is a concern for developers particularly in contexts where an analysis needs to be run often, e.g., in interactive development environments (IDEs).

1.2 Thesis Overview

Sound and precise analysis of JavaScript is unfortunately beyond the state of the art, and in this thesis, we explore the effectiveness of using *unsound* analysis to build tools to detect and remediate inefficiencies in asynchronous JavaScript programs. In the domain of finding inefficiencies, false negatives are tolerable as they correspond to missed optimization opportunities, and so an unsound analysis is appropriate in principle.

We explore the following thesis statement:

Unsound analysis of asynchronous JavaScript applications yields actionable insights and effective optimizations.

There are a few important parts to this statement. First, we investigate *unsound* analysis: sound and precise static analysis of JavaScript is elusive, so we employ methods that make no soundness guarantees. Next, *actionable insights*: we mean to devise techniques that communicate information to users that allows them to optimize their applications. Finally, *effective optimizations*: when possible, we design techniques that automatically repair code by transforming the application and optimizing it in some way. As an aside, when we say *optimize* we mean to improve some desirable and measurable aspect of code, e.g., its performance or size.

The thesis is organized as follows:

- Chapter 2 describes general background requisite for understanding this thesis. We describe the JavaScript language, how to build asynchronous JavaScript applications, how external code is imported into applications as well as the package management ecosystem (`npm`).
- Chapter 3 describes the literature broadly related to this thesis, and touches on program analysis of JavaScript (both static and dynamic), on code changes that will parallelize applications, on general refactoring, program understanding, and software debloating (relevant for Chapters 6 and 7).
- Chapters 4 through 7 describe four research projects wherein we developed approaches to optimize asynchronous JavaScript applications using unsound program analysis. Chapter 4 describes how general anti-patterns related to misuses of promises can be detected and effectively communicated to programmers, Chapter 5 describes an approach for detecting and refactoring misuses of ORMs in JavaScript applications, Chapter 6 describes an approach for leveraging unsound analysis to remove

dead code from applications, and Chapter 7 describes a situation where applications are refactored to lazily load packages used only in the context of event handlers.

- Finally, Chapter 8 concludes with an in-depth retrospective and discussion of how to best leverage unsound analysis.

Chapter 2

The JavaScript Language and Ecosystem

This chapter reviews JavaScript and its ecosystem. It includes an overview of the JavaScript event-loop architecture, presents callbacks, promises, and `async`/`await`, and discusses the many mechanisms for including external files and modules in JavaScript applications. Readers familiar with these concepts should feel free to skip this chapter.

2.1 JavaScript Language Primer

JavaScript is a *dynamically typed* language, meaning that programmers do not write type annotations. Beyond being dynamically typed, the JavaScript semantics are also extremely dynamic. Put simply, JavaScript is highly expressive, and the language rarely restricts programmers; among other things, programmers can execute dynamically constructed strings as code at run time, non-existent object properties can be safely accessed, and objects can be extended at runtime.

2.1.1 Executing Arbitrary Strings as Code

In JavaScript, the `eval` function takes a string as an argument and executes it as if it were JavaScript source code. Previous work by Richards et al. [183] investigated what programmers typically do when they use it, finding that `eval`'d strings exercise the full gamut of the language. Thus, such strings are a total wildcard from the point of view

of static analysis: if programmers build up strings dynamically and `eval` them, a sound static analysis would have to determine exactly what that string is, which would essentially amount to running a large part of the program. As such, analyses are typically pessimistic about the outcome of a call to `eval`. Moreover, use of `eval` is widespread. Richards et al. [184] investigate 103 web sites, and find that they *all* use `eval`, from a handful to a few hundred times per application. They also stress the variety of code that `eval` executes.

2.1.2 Dynamic Property Access and Extension

In JavaScript, programmers can use dynamic values to index and extend objects at runtime. To help illustrate, consider the following code snippet:

```
1  let O = {};
2  let b = "m";
3  O.m = () => { return 2; };
4
5  O.m();      // calls m
6  O["m"]();   // calls m
7  O[b]();     // calls m
```

Here, we first define an empty object `O` (line 1) and a variable `b` initialized with the string `"m"`. We then dynamically extend `O` with a property `m` initialized with an arrow function that returns the number `2` (line 3). Then, we illustrate three different ways to call `m`: (1) directly with the property name on line 5, (2) *dynamically* with the property name passed as a string value on line 6, and again (3) *dynamically* with a variable that evaluates to `"m"` on line 7. Now, imagine we want to compute the call graph for a JavaScript application. The above code snippet illustrates a particularly tricky combination of JavaScript semantics: functions are first-class values, meaning you can create them dynamically and pass them around, you can dynamically extend objects at runtime, and you can access object properties with computed values (you can also extend objects using computed values). To soundly determine all possible call targets of the expression `O[b]()`, a static analysis would need to determine all possible values of `b`, essentially running part of the program. A sound static analysis would either need to over-approximate call targets and be imprecise but scalable, or be precise at the cost of scalability.

Now, one wonders how often programmers actually use these dynamic features. According to a study by Richards et al. [184], JavaScript programs exhibit a high degree of dynamism and thus present a "harsh terrain for static analysis". For instance, JavaScript has fewer monomorphic call sites than Java (81% vs 90%), all projects studied had megamorphic call sites with over 32 possible call targets (2.5% of call sites had more than

5 potential targets), and most studied applications use `eval` and `eval`'d code strings are extremely varied.

On top of this basic complexity, JavaScript also has mechanisms for writing asynchronous programs (naturally, as the web is an asynchronous environment). This is introduced next.

2.2 Asynchronous JavaScript

There are three major ways to build asynchronous JavaScript applications, presented in § 2.2.2 and § 2.2.3. First, it helps to understand how asynchronous computations are realized in the language, discussed next.

2.2.1 The Event Loop

JavaScript only has a single user thread, but JavaScript applications rely heavily on I/O operations, e.g., interaction with servers and user input handling. To reconcile this, the language has run-time model based on an *event loop* that enables it to perform operations asynchronously despite being single-threaded. Essentially, the event loop is a queue of function calls (i.e., callbacks) to be executed, which follow run-to-completion semantics; calling functions asynchronously has the effect of loading them onto the event loop. Once on the event loop, a callback is executed similarly to any other synchronous code.

2.2.2 Callbacks

This style of asynchronous programming relies on functions being registered as *listener callbacks* for specific events, which are called when the associated event is emitted. As an example, consider the following code snippet, which declares a function `onClick` that is then registered as a listener callback handling the `"click"` event:

```
function onClick(event) { /* handler logic */ }
document.addEventListener("click", onClick);
```

The call `document.addEventListener("click", onClick)` registers `onClick` as the callback to handle the `"click"` event on the `document` component of the web page. Later, when a user clicks on the page, the `"click"` event fires and a call to `onClick` is placed on the event loop.

2.2.3 Promises and `async/await`

This section reviews promises [73, Section 27.2] and the `async/await` feature [73, Section 15.8] features, which were added to JavaScript in recent years to facilitate asynchronous programming.

A *promise* is an object that represents the value computed by an asynchronous computation, and is in one of three states: *pending*, *fulfilled*, or *rejected*. Upon construction, a promise is in the *pending* state. If the computation associated with a promise p successfully computes a value v, then p transitions to the *fulfilled* state, and we will say that p is *fulfilled with value* v. If an error e occurs during the computation associated with a promise p, then p transitions to the *rejected* state, and we will say that p is *rejected with value* e. The state of a promise can change at most once; accordingly, we will say that a promise is *settled* if it is *fulfilled* or *rejected*.

Creating promises. Promises can be created by invoking the `Promise` constructor, passing it an *executor function* expecting two arguments, `resolve` and `reject`, for fulfilling or rejecting the newly constructed promise, respectively. E.g., the following code snippet

```
let c = ...
let p1 = new Promise( (resolve, reject) => {
  if (c){ resolve(3) } else { reject("error!") }
})
```

assigns to `p1` a new promise that is fulfilled with the value `3`, or rejected with the value `"error!"`, depending on the value of `c`. The functions `Promise.resolve` and `Promise.reject` accommodate situations where a promise always needs to be fulfilled or rejected with a specified value, respectively. For example, the following code snippet:

```
let p2 = Promise.resolve(4)
let p3 = Promise.reject("error!")
```

assigns to `p2` and `p3` promises that are fulfilled with the value `4` and rejected with the value `"error!"`, respectively.

Reactions. To specify that a designated function should be executed asynchronously upon the settlement of a promise, programmers may register *reactions* on promises using methods `then` and `catch`. Here, a reaction is a function that takes one parameter, which is bound to the value that the promise was fulfilled or rejected with. For example, consider the following code snippet:

```
p2.then( (v) => console.log(v*v) )
```

This snippet extends the previous example by registering a reaction on the promise referenced by variable `p2` to print the value `16`[1]. Similarly, the following code snippet:

```
p3.catch( (e) => console.log("error: " + e) )
```

will cause the text "`error:  error!`" to be printed.

Promise chains. The `then` method returns a promise. If the reaction that is passed to it returns a (non-promise) value v, then this promise is fulfilled with v. If the reaction that is passed to it throws an exception e, then this promise is rejected with e. Furthermore, if `then` is used to register a reaction f on a promise p, then the rejection of p with a value e will cause the rejection of the promise returned by p`.then(` f`)` with the same value e. This enables the construction of *chains* of promises. In the following code snippet, a promise chain is created starting with variable `p1` as defined above:

```
p1.then( (v) => v+1 )
  .then( (w) => console.log(w) )
  .catch( (err) => console.log("an error occurred.") )
```

if `p1` was fulfilled with `3`, then the reaction `(v) => v+1` will be executed asynchronously with `v` bound to the value `3` and return the value `4`, so the promise created by this call to `then` is fulfilled with the value `4` as well. Since a reaction `(w) => console.log(w)` was registered on that promise, the value `4` will be printed. If, on the other hand, `p1` was rejected with the value `"error!"`, the promises created by both calls to `then` will be rejected as well, with the same value, causing the reaction on the last line to execute, which prints `"an error occurred."`.

Linked promises. So far, we have only considered situations where a function f that is registered as a reaction on a promise returns a non-promise value. However, if f returns a promise p, that promise becomes *linked* with the promise p' created by the call to `then` (or `catch`) that was used to register the reaction. Concretely, this means that p' will be fulfilled with a value v if/when p is fulfilled with v, and p' will be rejected with a value e if p is rejected with e, and if p remains pending then so will p'. Consider the following example:

```
let p4 = Promise.resolve(5);
let p5 = new Promise( (resolve,reject) =>
  setTimeout(() => resolve(6), 1000) )
p4.then( (v) => p5 )
  .then( (w) => console.log(w) ) // prints 6 after one second
```

[1]The `then` method optionally accepts a reject-reaction as its second argument.

Here, the promise referenced by `p4` is fulfilled with `5`, and the promise referenced by `p5` is fulfilled with `6` after 1000 milliseconds have elapsed. The reaction `(v) => p5` that is registered on `p4` returns `p5`, so the promise created by this call to `then` becomes linked with `p5`, i.e., it will be fulfilled with 6 after 1000 milliseconds have passed. The last line registers another reaction on this promise, so the value `6` is printed after 1000 milliseconds.

Synchronization. Several functions are provided for synchronization. The `Promise.all` function takes an array of promises `[` $p_1, \cdots, p_n$`]` as an argument and returns a promise that is either fulfilled with an array `[` $v_1, \cdots, v_n$`]` containing the values that these promises are fulfilled with, or that is rejected with a value e_i, if p_i is the first promise among $p_1, \cdots, p_n$ that is rejected, and e_i is the value that it is rejected with. Other synchronization functions include `Promise.race` and `Promise.any`. For example[2], the following snippet prints `Array [3, 42, "foo"] after 1 second`:

```
let p6 = Promise.resolve(3);
let p7 = 42;
let p8 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'foo');
});
Promise.all([p6, p7, p8])
  .then((vs) => console.log(vs); );
```

Promisification. Promisification is a mechanism for automatically adapting an asynchronous event-driven API into a promise-based API. It assumes that methods in an event-driven API meet two requirements: (i) the callback function is the last parameter, (ii) upon completion of the asynchronous operation, the callback function is invoked with two parameters `err` and `result`, where `err` is a value that indicates whether an error has occurred, and `result` contains the result of the asynchronous computation otherwise. In such cases, an equivalent promise-based API can be derived by creating a new promise that invokes the event-driven API, passing it a callback that rejects the promise with `err` if an error occurred, and fulfills it with `result` otherwise. Promisifying event-driven APIs can be done using the built-in `util.promisify` function.

`async/await`. JavaScript allows a function to be declared as `async` to indicate that it computes its result asynchronously. An `async` function f returns a promise: if no exceptions occur during the execution of f, this promise is fufilled with the returned value, and if

[2] Adapted from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all.

an exception e is thrown, then the promise is rejected with e. Inside the body of `async` functions, `await`-expressions may be used to await the settlement of promises, including promises created by calls to other `async` functions. Concretely, when execution encounters an expression `await` x during the execution of an `async` function, control returns to the main event loop. At some later time, when the promise that x evaluates to has settled, execution resumes. If that promise was fulfilled with a value v, then execution resumes with the entire `await`-expression evaluating to v. If the promise was rejected with a value e, then execution resumes with the entire `await`-expression throwing an exception e.

The `async`/`await` feature has been designed to interoperate with promises, as is illustrated by the example below.

```
33  import fs from 'fs'
34  async function analyzeDir(dName){
35    let fNames = await fs.promises.readdir(dName);
36    let ps = fNames.map( (fName) => fs.promises.stat(fName) );
37    let fStats = await Promise.all(ps);
38    let sum = fStats.reduce((acc,v) => acc + v.size, 0);
39    console.log(sum);
40  }
```

The example shows an `async` function `analyzeDir` that prints the sum of the sizes of the files in the directory identified by its parameter `dName`. On line 35, an await-expression is used to await the results of the built-in `readdir` operation; this operation returns a promise that is eventually fulfilled with an array containing the names of files in the specified directory, which is assigned to `fNames`. On line 36, the `map` operation on arrays is used to map the built-in `fs.stat` operation[3] over this array, resulting in an array `ps` of promises that will eventually resolve to objects containing meta-information for each file. `Promise.all` is used on line 37 to create a promise that is eventually fulfilled with the meta-information objects for each of the files, and an await-expression is used to await this result so that it can be stored in a variable `fStats`. On line 38, the `reduce` operation on arrays is used to compute the sum of the sizes of the files, and this sum is printed on line 39.

JavaScript's async/await feature can be thought of as syntactic sugar for promise-based asynchrony. Consider:

```
41  function fetchAsynchronously(url) {
42    fetch(url)
43    .then(response => response.json())
44    .then(jsonResponse => {
45      // do something
46    });
47  }
```

[3] `fs.stat` is a library function that returns an object that contains various information about a file, including its size; see https://nodejs.org/api/fs.html#fs_class_fs_stats.

Here, the function `fetchAsynchronously` takes a `url`, fetches it, converts it to JSON, and then does something with it—all using promises. In this setup, the bulk of the function logic would be in the body of the last callback (`// do something`). Using `async`/`await`, we can write the function more concisely as:

```
48  async function fetchAsynchronously(url) {
49    const response = await fetch(url);
50    const jsonResponse = await response.json();
51    // do something
52  }
```

2.3 The npm Package Manager

JavaScript developers enjoy `npm`, a thriving ecosystem of over two million external packages [170]. To include external code in their project, a developer simply needs to open a command line interface (CLI), type `npm install p-name`, and the code for package `p-name` and all of its dependencies will be downloaded. Once downloaded, programmers can import the package using `require`, or the static `import` statement and dynamic `import` function.

2.3.1 JavaScript Import Mechanisms

require The traditional method of including external code in JavaScript is to use `require`, a function that dynamically *and synchronously* loads and executes the package matching the supplied name. Consider:

```
const xlsx = require("xlsx");
function importXLSXData(data) {
  const contents = xlsx.read(data, {...});
  // do stuff with the contents.
}
```

First, the `"xlsx"` package is imported at runtime and saved in the `xlsx` global variable. `"xlsx"` exports a `read` function to convert raw spreadsheet data, and so inside `importXLSXData` the exported function is referenced as a property on the `xlsx` object (`xlsx.read`). Notably, `xlsx` contains *the entire* package code.

static import ECMAScript 6 introduced the static `import` declaration as an alternative to the dynamic `require`. These `import` statements must be at the top level, all bindings must be identifiers, and the package name must be a string literal (this makes them easier

to analyze statically); e.g., the statement `import * as xlsx from "xlsx"` imports the entire `"xlsx"` package. A major advantage of static `import` statements is that a developer can specify which parts of a package they want to import; e.g., in the following snippet, the `read` function exported by `"xlsx"` is imported directly:

```
import { read } from "xlsx";
function importXLSXData(data) {
  const contents = read(data, {...});
  // do stuff with the contents.
}
```

The strict nature of these static import statements allows static analyzers to more effectively determine the extent to which an application exercises the code it imports, which can sometimes lead to smaller distributions—this is called *tree-shaking* [186, 232]. Unfortunately, JavaScript's high degree of dynamism limits the power of these static analyses [175, 125, 130], preventing tree-shaking from removing much code.

dynamic import Static imports are syntactically rigid by design, and so ECMAScript 2020 introduced a dynamic, *asynchronous* `import` function. The `import` function accepts a string containing the name or path of a package as an argument and returns a promise. That promise can either resolve with an object containing all the exported functions and objects, or be rejected if the package cannot be found. This syntax is especially useful for importing large or rarely used external packages, since they will not be bundled with the rest of the application. This can often result in smaller initial application sizes and potentially faster load times. The following code snippet illustrates how to dynamically import `"xlsx"` only in the context of `importXLSXData`:

```
async function importXLSXData() {
    const xlsx = await import("xlsx");
    const data = xlsx.read(...);
}
```

Note that if a dynamic import for a particular package is encountered more than once, the package is loaded only once, and all subsequent invocations resolve to the same cached instance. Thus, even if `import("xlsx")` or `importXLSXData` is invoked multiple times, the `"xlsx"` package will be loaded only once and served to all subsequent invocations.

Chapter 3

Related Work

In this chapter, we review the literature as it applies to this thesis, which touches on program analysis, debloating and reducing program size, optimizing programs, understanding programs, and refactoring. This chapter presents a general literature overview, while each of Chapters 4-7 have their own "Relation to Previous Literature" section that refers back to this chapter, and also describes some related work specific to the chapter itself.

3.1 Analysis of JavaScript

An essential component of this thesis is *program analysis*. There are two prevailing methods to analyze programs: (1) *static analysis*, which analyzes source code without running the actual program, and (2) *dynamic analysis*, which must execute the code to perform its analysis. There are advantages and disadvantages to each approach. While dynamic analysis yields precise insights about programs (since the program actually ran), it is unclear how well insights about one particular execution generalize, and is limited in that code needs to be run, which can be surprisingly difficult. In contrast, static analysis can simply analyze the source code, but must model any inaccessible code (e.g., some library code, or opaque language functionality like JavaScript's native models), and importantly must account for all possible program behavior to preserve soundness. This last point is of particular concern in dynamic languages where the behavior of a program at runtime can be unpredictable. In a sense, dynamic analysis under-approximates the behavior of a program by considering a finite number of executions of the program, while static analysis over-approximates behavior through it's model of potential program executions.

Broadly, dynamic languages pose many unique challenges to the design of language tooling, which often relies on program analysis. Work by Wei et al. [234] present a fuzzing approach for deep-learning libraries, and in their implementation of this as a tool for Python have struggled with the dynamism inherent to the language (to tackle the issue, they performed a corpus analysis to infer type specifications for functions). Also in the context of Python, Yang et al. [243] present a study of how the complex features of Python are used, and how frequently, culminating in a minimum set of features that any practical static analysis of the language should support. A separate study [177] describes the common language features across multiple Python projects. As for JavaScript, Richards et al. [183] show that `eval` is everywhere in JavaScript, and so any language tools need to be aware of the unsoundness.

3.1.1 Static Analysis

There are several static analysis frameworks for JavaScript. `JSAI` [122] is an abstract interpretation framework for JavaScript, with user-configurable analysis sensitivity. `TAJS` [116] is a framework for inferring sound type information about JavaScript programs. `WALA` [227] is a set of libraries provided by IBM for analysis of Java bytecode and also JavaScript. CodeQL [152] is a framework for the declarative specification of static analyses as queries over an AST, data flow graph, and/or control flow graph of a program. Jelly [158] is a recent addition to the landscape, which is intentionally not fully sound, and who's design is based on several academic JavaScript analysis approaches (JAM [168], TAPIR [159], and ACG [90]).

Besides these tools, there are a number of other academic efforts to improve static analysis of JavaScript. Work by Madsen et al. [141] sheds light on how to leverage how a library is used to improve the precision of static analysis in JavaScript. Other work by Madsen et al. [142] discuss an approach to constructing an event-based call graph to find bugs in event-driven JavaScript applications. The aforementioned Jelly [158] tool incorporates several advancements in the field, incorporating: JAM [168], an approach to building call graphs that leverages the module structure of Node.js applications; TAPIR [159], an approach which localizes static analysis to user-specified code locations; and ACG [90], an approach that is intentionally unsound (e.g., it does not analyze dynamic property accesses) to improve scalability of static analyses.

In this thesis, we often leverage static analysis to detect inefficiencies, rather than prove properties about programs; in their paper, Gorogiannis et al. [103] propose a *True Positives Theorem* for establishing the "soundness" of static analyses as bug finding and testing tools,

rather than verification tools. Essentially, they propose a theorem to verify that a static analysis reports *no false positives*—these analyses may not be sound in the traditional verification sense, but this is nonetheless a useful and desirable property. This is directly applicable in the context of this work, which leverages unsound analysis to essentially build optimization tools, though we make no claim that no false positives are reported by these approaches. Also, in Chapters 5-7, we apply transformations to remediate detected inefficiencies, which is outside the scope of the theorem presented in the paper.

3.1.2 Dynamic Analysis

There are a few major paradigms in the space of dynamic analysis: instrumentation, and source code rewriting. With instrumentation, the runtime performs the analysis, and often exposes *hooks* for custom dynamic analyses to take advantage of. For example, the `Async Hooks` [3] API in Node.js exposes callbacks for major events in the lifetime of JavaScript promises (e.g., when they are created, destroyed, resolved, etc.) that provide precise information about runtime promises. More generally, the NodeProf [204] dynamic analysis framework in GraalVM [237, 235] exposes many callbacks for general program execution, e.g., function entry and exit, property reads and writes, etc. Instrumentation is relatively performant compared with source code rewriting, but suffers in that analyses are limited to the framework they are built in: an analysis written in NodeProf will not run on Node.js.

With source code rewriting, the program source code itself is transformed and statements are inserted collecting the desired information. A popular framework for this in JavaScript is Jalangi [194], which inserts callback functions with user-defined bodies into the relevant source code locations, but support for Jalangi has ceased and the alternative Jalangi2 [191] has not been updated in several years. Source code rewriting is more portable, but can significantly degrade the performance of analyzed applications. Moreover, source code rewriting approaches modify the program being analyzed, which can lead to behavioral differences between executions of a program with or without analysis code inserted.

Approaches using dynamic analysis are common. For example, work by Adamsen et al. [35] propose an approach using dynamic analysis to detect event races when web pages are initialized. Karim et al. [121] describe a platform-independent approach to dynamic taint analysis of JavaScript applications by generating stack machine instructions from program executions. Augur [39] is a tool that extends this approach, implementing it in the NodeProf framework and finding significant performance improvements over the original

approach that used Jalangi. Kreindl et al. [128] describe a language-agnostic framework for specifying taint labels for dynamic taint analysis, implementing it in `TruffleTaint` in the context of GraalVM.

3.1.3 Combining Static and Dynamic Analysis

Static and dynamic analysis are not mutually exclusive. Conceptually, insights from static analysis could be given as input to a dynamic analysis, or vice versa. Tzermias et al. [220] present a mixed approach to detecting security vulnerabilities in PDF documents, leveraging static page analysis and dynamic code execution. Lindorfer et al. [136] present a tool for classifying Android apps according to their percieved level of maliciousness using machine learning, and static and dynamic analysis. Balzarotti et al. [54] present SANER, a tool for sanitizing web applications that also combines static and dynamic analysis. Park et al. [176] describe an approach which uses dynamic analysis as a shortcut to speed up static analysis of JavaScript. Toman and Grossman [212] combine concrete and abstract interpretation to analyze programs that make extensive use of third party libraries and otherwise inaccessible code. Godefroid et al. [99] present DART, a fully automated testing framework that combines static analysis to determine the API of an application with dynamic analysis of generated random tests to analyze the behavior of the application.

3.2 Refactoring

This thesis discusses source code transformations (known as *refactoring*) that optimize application performance, and there is a wealth of work on this topic. Traditionally, *refactoring* refers to code transformations that preserve the behavior of the program, but improve the quality of the code, described in Fowler's seminal books [91, 93]. In this realm, Di Nucci et al. [68] explored if machine learning is feasible to use in the context of refactoring, Kamiya et al. [120] explored tokenizing source code to find code clones, Kessentini et al. [123] propose a parallel consensus-based method for detecting code smells efficiently. Work by Arteca et al. [49] present an automated refactoring for applications which use `async`/`await` by hoisting promise creation as early as possible, and delaying the `await`-ing of those promises as much as possible. Feldthaus et al. [89] describe a framework for specifying and implementing refactorings for JavaScript that uses pointer analysis.

There is also a wealth of work on refactoring applications to make them more asynchronous. Dig [69] present a toolset for many refactorings that increase parallelism. Desyn-

chronizer [100] refactors JavaScript applications to use asynchronous APIs where synchronous APIs were once used. Schäfer et al. [192] helps programmers refactor their code to take advantage of ReentrantLocks and ReadWriteLocks. Lin et al. [134] present a study to understand the use of `AsyncTask` in Android applications, and an accompanying tool to assist programmers in refactoring their applications to make use of it. Okur et al. [172] convert concurrent code from low-level abstractions to higher-level equivalents. Other research loosely in this space includes work by Khatchadourian et al. [124] on automatically parallelizing Java 8 streams, by Dig et al. [71] to parallelize Java loops, by Wloka et al. [236] on refactoring applications to be reentrant, by Dig et al. [70] for leveraging concurrency APIs to transform sequential code.

As for refactorings pertaining to database-backed applications, existing work has considered refactoring database schemas to improve performance. Ambler and Sadalage [44] catalogue *database refactorings*, i.e., behavior-preserving changes to a database schema such as moving a column from one table to another. Similarly, Xie et al. [238] and Wang et al. [230] study how application code must be updated in response to schema changes. Rahmani et al. [179] present an approach for avoiding serializability violations in database applications by transforming a program's data layout. These are primarily relevant to Chapter 5.

There are a number of non-academic tools for detecting and fixing "smells" in JavaScript applications: ESLint [12], JSLint [20], and JSHint [2]. These are most closely related to Chapter 4 and the anti-pattern detection, and indeed ESLint detects a few of the anti-patterns that we specified in the tool. That said, CodeQL allows for more precise static analyses (e.g., CodeQL supports data flow analyses, which ESLint does not), and many of the anti-patterns discussed in Chapter 4 are detected thanks to data flow.

3.3 Program Understanding

This thesis discusses code transformations that are achieved through unsound program analysis, and these changes are presented to developers so that they can study and carefully vet them before applying them. It is important that changes are understood by programmers, which is related to the study of *program understanding*. In the space of understanding asynchronous JavaScript programs, work by Alimadadi et al. [42] explores how event-based asynchrony in JavaScript can be better understood, how asynchrony on the entire application stack can be understood [41], and how DOM-sensitive changes affect program understanding [40]. More broadly, there have been experiments to determine the benefit of dynamic profile information for program comprehension tasks [62], a survey of

dynamic analysis techniques for program comprehension [63], and on understanding Ajax programs by connecting client- and server-side execution traces [251].

3.4 Debloating

Chapters 6 and 7 are concerned with reducing the size of JavaScript applications, known as software debloating. Many applications contain far more code than is required, commonly referred to as "dead code", and the study of debloating is the study of how to determine and safely remove this dead code. Besides increasing application size, dead code is undesirable as it increases the "attack surface" of an application, i.e., more code provides more opportunities for an attacker to take advantage of a system.

There is a wealth of work in this space. For example, Bhattacharya et al. [55] studies situations where functions accumulate more features than are strictly necessary, yielding poor performance when spurious functionality is not needed. Koo et al. [127] propose configuration-driven software debloating, where application configurations are linked with feature-specific libraries, and libraries are only loaded when the appropriate configuration criteria are met. This is a semi-automated process, and the code itself is not changed. Soto et al. [198] propose an approach to automatically specialize Java dependencies according to how they are used by the application's test suite, and Sharif et al. [195] propose a technique that leverages constant value configuration data to specialize applications. [37] present an type-inference based application extractor for Self [36] which extracts a bloat-free source file for distribution. The Jax application extractor for Java [209] relies on efficient type-based call graph construction algorithms such as RTA [53] and XTA [210] to detect unreachable methods, and further relies on a specification language [206] in which users specify classes and methods that are accessed reflectively, going above-and-beyond dead code elimination with, e.g., class hierarchy compaction [211]. Rayside and Kontogiannis [181] present a tool for extracting subsets of Java libraries using Class Hierarchy Analysis [67] to identify the subset of a library that is required by a specific application, though their work does not consider unsoundness. [127] present a technique relying on manual analysis of configuration files and profiling to obtain coverage information for executions in different configurations, minimizing based on that coverage.

Some recent work has been concerned with debloating JavaScript applications. Malavolta et al. [143] propose a technique to debloat client-side JavaScript applications with various levels of optimization; first, dead code is determined by consulting a call graph of the application, and one of the optimization levels proposed in the work replaces dead code with snippets to load the code lazily. Vasquez et al. [224] propose a technique that flags

external library functions as being potentially dead, and removes them once a programmer confirms that they are truly unused. Mininode [126] is a tool for debloating JavaScript applications using static analysis, and code can be removed at one of two levels of granularity: "coarse", where entire modules are removed, or "fine", where individual functions are removed. We discuss Mininode in more detail in Chapter 6 where it is most relevant.

In certain situations, outright removal of code is not desirable and instead developers want to load optional functionality on-demand; this is known as *code splitting*. In the space of *identifying* optional functionality that could be split, there is work [55] proposing an approach relying on a combination of human input, dynamic analysis, and static analysis to identify optional functionality. As for actually splitting the code, Doloto [138] proposes an approach that leverages developer-supplied application traces to automatically refactor applications to load entire "routes" lazily, only when they are needed; their approach performs dynamic loading synchronously, which is disallowed in the modern web standard. [129] proposed a code-splitting technique for Java that partitions classes into separate "hot" and "cold" classes to avoid transferring code that is rarely used. [225] present an optimistic compaction technique for Java applications, where minimized distributions are outfitted with a custom class loader that performs partial loading and on-demand code addition.

Developers are also interested in minimizing code size, particularly when preparing production-level distributions of their applications. Several implementations of Smalltalk developed in the 1990s (e.g., [174, 112]) include features for "packaging" or "delivering" applications, and IBM's 1997 Handbook for VisualAge for Smalltalk [112] describes a reference-following strategy to determine minimal code for a package. Compacting code is a related area, for example [66] present Squeeze++, a link-time code compactor for low-level C/C++ code. Another facet of this area is specializing distributions: [195] present TRIMMER, which specializes LLVM bytecode applications to their deployment context using input specialization. The performance impact of using application bundles has also been studied in the context of Java, where [110] study performance issues that arise when bundles of JVM class files for Java applications are downloaded from a server.

3.5 Conclusion

By now, we have reviewed the vast landscape of literature and have given the background knowledge required for understanding, broadly, the contents of this thesis. We are aiming to show that *unsound analysis of asynchronous JavaScript applications yields actionable in-*

sights and effective optimizations, and the next four chapters present exemplar approaches in support of this statement. The chapters are:

- Chapter 4 describes how general anti-patterns related to misuses of promises can be detected and effectively communicated to programmers;
- Chapter 5 describes an approach for detecting and refactoring misuses of ORMs in JavaScript applications;
- Chapter 6 describes an approach for leveraging unsound analysis to remove dead code from applications;
- and Chapter 7 describes a situation where applications are refactored to lazily load packages used only in the context of event handlers.

Chapter 4

Anti-Pattern Identification

Abstract

Promises and async/await have become popular mechanisms for implementing asynchronous computations in JavaScript, but despite their popularity, programmers have difficulty using them. In this chapter, we identify 8 anti-patterns in promise-based JavaScript code that are prevalent across popular JavaScript repositories. We present a light-weight static analysis for automatically detecting these anti-patterns. This analysis is embedded in an interactive visualization tool that additionally relies on dynamic analysis to visualize promise lifetimes and instances of anti-patterns executed at run time. By enabling the user to navigate between promises in the visualization and the source code fragments that they originate from, problems and optimization opportunities can be identified.

We implement this approach in a tool called *DrAsync*, and found 2.6K static instances of anti-patterns in 20 popular JavaScript repositories. Upon examination of a subset of these, we found that the majority of problematic code reported by *DrAsync* could be eliminated through refactoring. Further investigation revealed that, in a few cases, the elimination of anti-patterns reduced the time needed to execute the refactored code fragments. Moreover, *DrAsync*'s visualization of promise lifetimes and relationships provides additional insight into the execution behavior of asynchronous programs and helped identify further optimization opportunities.

4.1 Introduction

The `async`/`await` feature [73, Section 15.8] was added to the JavaScript programming language in 2017 to facilitate asynchronous programming with convenient syntax and error handling. Programmers can designate a function as `async` to indicate that it performs an asynchronous computation, and `await`-expressions may be used in these functions to await the result of other asynchronous computations. The JavaScript community has enthusiastically embraced this feature, as it is less error-prone than event-driven programming and syntactically much less cumbersome than the promises feature [73, Section 27.2] on which it builds. However, many JavaScript programmers are still unfamiliar with asynchronous programming, and particularly with `async`/`await` and how it interacts with promises. As a result, they sometimes produce code creating redundant promises, or code that performs poorly because the ordering of asynchronous computations is constrained unnecessarily [49].

We identify 8 anti-patterns involving the use of promises and `async`/`await` that commonly occur in JavaScript programs. These anti-patterns reflect designs that are likely to be suboptimal because they may create promises unnecessarily, perform synchronization that is redundant, or cause code to become needlessly complicated. Examples of these anti-patterns include redundant uses of `await`, the use of `await` in loops over arrays, and explicit creation of new promises where none are needed. In many cases, these anti-patterns can be refactored into code that is more concise or more efficient.

We developed a lightweight static analysis to detect these anti-patterns directly in source code, and implemented this analysis as a set of CodeQL queries [51, 5]. Furthermore, to help programmers understand the run-time impact of the anti-patterns, we developed *DrAsync*, a profiling tool that visualizes the lifetime of the promises created by an application, and that highlights the run-time instances of each anti-pattern. This enables programmers to focus their attention on anti-patterns in frequently-executed code and provides valuable insights into where performance bottlenecks occur.

In an experimental evaluation, *DrAsync*'s static analysis detected 2.6K instances of anti-patterns in 20 JavaScript applications, and *DrAsync*'s dynamic analysis determined that, in the aggregate, these anti-patterns were executed 24K times by the application test suites. To evaluate whether the detected anti-patterns represent actionable findings, we selected 10 instances of each anti-pattern randomly and attempted to manually refactor them to eliminate the anti-pattern. We were able to successfully refactor 65 of these 80 instances, and determined that, in certain cases, these refactorings can have measurable impact on the number of promises created by an application, or the time needed the execute affected code fragments.

In summary, this chapter contains:

- the definition of 8 anti-patterns that commonly occur in asynchronous JavaScript code;
- *DrAsync*, a tool that relies on static and dynamic program analysis to detect anti-patterns and visualize promises and occurrences of anti-patterns during program execution, enabling programmers to quickly identify quality issues and performance bottlenecks;
- an empirical study of 20 JavaScript applications in which *DrAsync* is used to identify 2.6K anti-patterns which are executed 24K times, confirming that they are pervasive; and
- a case study that investigates whether 10 randomly chosen instances of each anti-pattern can be refactored, providing evidence that the majority of anti-patterns reported by *DrAsync* can be eliminated through refactoring. Further analysis of these results suggests that, under certain conditions, eliminating anti-patterns may improve performance.

The remainder of this chapter is organized as follows: § 4.2 points readers to relevant background discussed in Chapter 2, § 4.3 showcases many pitfalls related to using promises in JavaScript, § 4.4 formally describes the anti-patterns, § 4.5 describes the static analyses, dynamic profiler, and visualization that make up the *DrAsync* tool, § 4.6 describes selected case studies in refactoring issues, § 4.7 presents an evaluation of the tool, § 4.8 discusses threats to validity of this work, § 4.9 positions this work in the context of the literature, § 4.10 concludes and § 4.11 presents a short retrospective of this work, and puts it into context with respect to the other work in this thesis.

4.2 Promises and `async`/`await`

This chapter describes anti-patterns related to misuse of promises and `async`/`await`. Please refer to Section 2.2.3 for relevant background on JavaScript promises.

4.3 Motivating Examples

Asynchronous programming is rife with pitfalls. As a first example, consider SAP's *ui5-builder* project, which provides modules for building UI5 projects. *ui5-builder*'s file

`ResourcePool.js` contains the following function, which *DrAsync* flagged as an instance of the *promiseResolveThen* anti-pattern that will be presented in Section 4.4:

```
67  async getModuleInfo(name) {
68    let info = this._dependencyInfos.get(name);
69    if (info == null) {
70      info = Promise.resolve().then(async () => {
71        const resource = await this.findResource(name);
72        return determineDependencyInfo(resource, ... );
73      });
74      this._dependencyInfos.set(name, info);
75    }
76    return info;
77  }
```

On line 70, `Promise.resolve()` is invoked to create a promise that is fulfilled immediately with the value `undefined`[1]. On the same line, an `async` function is registered as a fulfill reaction on this promise, so this reaction is asynchronously invoked with `undefined` as an argument. This means that 3 promises are created when the reaction executes: (i) the promise created by `Promise.resolve`, (ii) the promise created by the invocation of `then`, and (iii) the promise created by the invocation of the `async` function. This is manifested in *DrAsync*'s visualization as an extremely short-lived promise linked two other, longer-running promises (see Figure 4.1).

In this case, the code can be refactored as such:

```
78  async getModuleInfo(name) {
79    let info = this._dependencyInfos.get(name);
80    if (info == null) {
81      info = (async () => {
82        const resource = await this.findResource(name);
83        return determineDependencyInfo(resource, ... );
84      })();
85      this._dependencyInfos.set(name, info);
86    }
87    return info;
88  }
```

Now, only one promise is created (on line 81, by invoking the `async` function). This code is executed 204 times in *ui5-builder*'s test suite, and 2 fewer promises are executed each time. Besides being more efficient, the code is more concise, and easier to understand.

As another example, consider `appcenter-cli`, developed by Microsoft, which implements the Command Line Interface (CLI) for the Visual Studio Code (VSCode) Interactive Development Environment (IDE). Function `cpDir`, defined on lines 89-94 in `src/util/misc/promisified-fs.ts`, implements the copying of a directory:

[1]Since no argument is passed in the call to `Promise.resolve`, the value `undefined` is used by default.

Figure 4.1: An example of the *promiseResolveThen* anti-pattern found in `getModuleInfo`. The user selected one of the promises in a promise chain originating from an empty `Promise.resolve()`, identified by Label A, and the reaction's promise is shown with Label B, and finally the promise belonging to the async function is shown with Label C.

```
89  async function cpDir(source, target) {
90    // details omitted
91    const files = await readdir(source);
92    for (let i = 0; i < files.length; i++) {
93      const sourceEntry = path.join(source, files[i]);
94      const targetEntry = path.join(target, files[i]);
95      await cp(sourceEntry, targetEntry);
96    }
97  }
```

This code reads the source directory `source` on line 91 and then iterates over the resulting list of file names. In each iteration of the loop, a call to function `cp` is `await`-ed, which copies a file from `sourceEntry` to `targetEntry`. Here, `cp` returns a promise that is fulfilled once `sourceEntry` is successfully copied to `targetEntry`, or rejected if an error occurs. It is important to note that this use of `await` in a loop causes the execution of function `cpDir` to be paused until the promise returned by `cp` is fulfilled, and execution will pass back to the main event loop at this time so that other event handlers can be executed in the meantime. This is manifest in *DrAsync*'s visualization by a "staircase" pattern of promises that have lifetimes that do not overlap (see Figure 4.2).

In this case, the copying of file-entries need not be sequential, and we can refactor the above code as follows:

```
98  async function cpDir(source, target) {
99    // details omitted
100   const files = await readdir(source);
101   await Promise.all(files.map(file => {
102     const sourceEntry = path.join(source, file);
103     const targetEntry = path.join(target, file);
104     return cp(sourceEntry, targetEntry);
105   }));
106 }
```

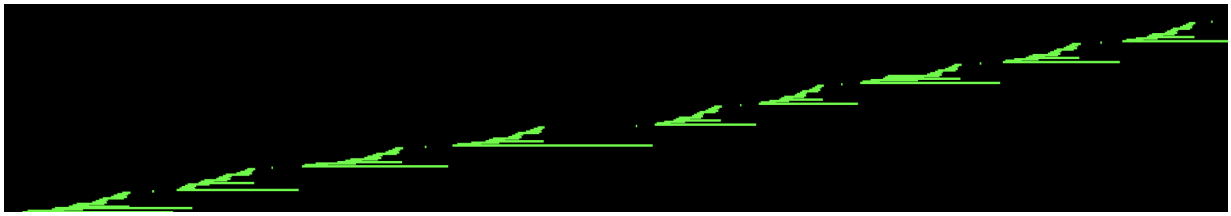


Figure 4.2: An example of the *loopOverArrayWithAwait* anti-pattern in the visualization, from a view depicting an overview of all promises. Each loop iteration is clearly separated, with no overlapping promises.

Here, we turn the for-loop into a map over the `files` array, mapping a function that returns the promise associated with `cp`. We then `await` the entire array of promises with `Promise.all` (line 101), which will wait for all these promises to resolve. This refactoring preserves the behavior of *appcenter-cli*'s tests, and enables additional concurrency because, although JavaScript is single-threaded at the language level, it relies on I/O libraries that can execute concurrently [49]. We will report in Section 4.7 how the refactoring significantly improves the performance of the loop.

These anti-patterns are detected using a simple static analysis. Our *DrAsync* tool additionally relies on dynamic analysis to determine how often each instance of an anti-pattern is executed, and helps programmers prioritize which code should be fixed. For instance, we found many instances of the "await-in-loop" pattern in *appcenter-cli*, but the highlighted `cpDir` example was by far the most frequently executed while running the application's tests.

4.4 Anti-Patterns

This section defines a set of anti-patterns that occur frequently in asynchronous JavaScript applications. We identified most of these through manually inspecting JavaScript source code[2], and inspecting visual profiles produced by *DrAsync* for noteworthy patterns (e.g., repetitive structures or promises that are very short-lived). In addition, a search for issues related to promises and `async`/`await` on the popular stackoverflow forum turned up the *explicitPromiseConstructor* [3] and *customPromisification* [4] anti-patterns.

[2]Section 4.7.1 provides further detail on the process for selecting subject applications.
[3]https://stackoverflow.com/questions/23803743
[4]https://www.grouparoo.com/blog/promisifying-node-functions

$$
\begin{aligned}
\mathit{asyncFunctionNoAwait} &= \{\ f \mid f\ \mathit{async} \wedge (\exists e_0, e_1 : e_0 = \texttt{await}\ e_1 \Rightarrow e_0\ \not\triangleleft f)\ \} \\
\mathit{asyncFunctionAwaitedReturn} &= \{\ f \mid f\ \mathit{async} \wedge (\exists e_0, e_1 : e_0 = \texttt{return}\ e_1 \wedge e_0 \triangleleft f) \Rightarrow \exists e_2 : e_1 = \texttt{await}\ e_2\ \} \\
\mathit{loopOverArrayWithAwait} &= \{s_0 \mid \exists e_0, e_1, e_2, e_3, s_1 : s_0 = \texttt{for}(e_0, e_1, e_2)\{s_1\} \wedge \textbf{isArrayTest}(e_1) \wedge \texttt{await}\ e_3 \triangleleft s_1\ \} \\
\mathit{promiseResolveThen} &= \{e_0 \mid \exists e_1, f : e_0 = \texttt{Promise.resolve}(e_1).\texttt{then}(f)\ \} \\
\mathit{executorOneArgUsed} &= \{e_0 \mid e_0 = \exists f, v_0, v_1 : \texttt{new Promise}(f) \wedge v_0 = \textbf{arg}(f, 0)\ \wedge v_1 = \textbf{arg}(f, 1) \wedge \\
&\qquad (\exists e_1, e_2 : e_1, e_2 \triangleleft f \wedge e_1, e_2 \in \{v_0, v_1\} \Rightarrow e_1 = e_2)\} \\
\mathit{reactionReturnsPromise} &= \{e_0 \mid \exists e_1, e_2, f : e_0 = e_1.\texttt{then}(f) \wedge \texttt{return}\ e_2 \triangleleft f \wedge \\
&\qquad (e_2 = \texttt{Promise.resolve}(\cdots) \vee e_2 = \texttt{Promise.reject}(\cdots))\ \} \\
\mathit{customPromisification} &= \{e_0 \mid \exists f_0, f_1, f_2, s_0, s_1, v_0, v_1 : e_0 = \texttt{new Promise}(f_0) \wedge f_1(..., f_2) \triangleleft f_0 \wedge \\
&\qquad \texttt{if}\ (\cdots)\ \{\texttt{s}_0\}\ \texttt{else}\ \{\texttt{s}_1\} \triangleleft f_2\ \wedge\ v_0 = \textbf{arg}(f_0, 0)\ \wedge v_1 = \textbf{arg}(f_0, 1) \wedge \\
&\qquad ((v_0 \triangleleft s_0 \wedge v_1 \triangleleft s_1) \vee (v_0 \triangleleft s_1 \wedge v_1 \triangleleft s_0))\ \} \\
\mathit{explicitPromiseConstructor} &= \{e_0 \mid \exists e_1, f_0, f_1, f_2, v_0, v_1, v_2, v_3 : e_0 = \texttt{new Promise}(f_0) \wedge e_1.\texttt{then}(f_1).\texttt{catch}(f_2) \triangleleft f_0 \wedge \\
&\qquad v_0 = \textbf{arg}(f_0, 0)\ \wedge\ v_1 = \textbf{arg}(f_0, 1)\ \wedge v_2 = \textbf{arg}(f_1, 0)\ \wedge v_0(v_2) \triangleleft f_1 \wedge \\
&\qquad v_3 = \textbf{arg}(f_2, 0)\ \wedge v_1(v_3) \triangleleft f_2\ \}
\end{aligned}
$$

Figure 4.3: Anti-patterns that commonly occur in asynchronous JavaScript code.

It is important to note that an occurrence of one of these anti-patterns is not necessarily a reflection that a design is "wrong" or "inefficient", but it indicates that it is likely that the code can be improved to make it more efficient by creating fewer promises or enabling additional parallelism, or to make it more concise. Section 4.6 presents a case study that investigates, for a representative subset of instances of these anti-patterns, how often we were able to refactor them manually. Section 4.7 presents an empirical evaluation that reports on the prevalence of each of the anti-patterns.

Figure 4.3 defines each anti-pattern as a set of AST nodes that meet some specified criteria. In the figure, we use f to represent functions (including arrow functions and class methods), e to represent expressions, and s to represent statements. Subscripts are used in cases where a predicate refers to multiple program elements of the same kind. Furthermore, f *async* denotes that f is an `async` function, and $e \triangleleft f$ (read as: "f contains expression e") indicates that f is the innermost function declaration or function expression such that e syntactically occurs within the body of function f.

asyncFunctionNoAwait. This anti-pattern is defined as any function f such that: (i) f is an `async` function and (ii) for any expression $e_0 =$ `await` e_1, e_0 does not occur in the body of f. In other words, the pattern identifies `async` functions that do not contain any `await` expressions. As we will discuss in Section 4.6, such functions can often be refactored into functions that are not `async`, to avoid the creation of a promise each time the function is executed. Note that the scope of this refactoring may expand beyond f itself: functions calling f may no longer need to await the result of the call f.

asyncFunctionAwaitedReturn. This anti-pattern is defined as any function f such that: (i) f is an `async` function and (ii) any return-expression in f is an await-expression. In such cases, the use of `await` is redundant, because the value v that the await-expression evaluates to is immediately used to settle the promise created by the `async` function (which itself would need to be awaited—it is more efficient to return the promise as it will become linked with the promise created by the `async` function).

loopOverArrayWithAwait. This anti-pattern covers `for`-loops like `for`$(e_0, e_1, e_2)\{s_1\}$ where (i) the condition e_1 tests that the loop iterates over an array by checking that it refers to the `Array.prototype.length` property (using auxiliary function **isArrayTest**), and (ii) the body s_1 of the loop contains at least one `await`-expression. This situation is well-known in the JavaScript community as being needlessly inefficient in situations where the iterations of the loop are independent of one another, and the ESLint checker [12] has a rule for detecting it. As we will discuss in Section 4.6, in many cases, such loops can be refactored to use `Promise.all` and `Array.prototype.forEach` to enable additional parallelism.

promiseResolveThen. An expression e_0 = `Promise.resolve(`$\texttt{e}_1$`) .then(`f`)` is constructed, i.e., a new promise is constructed on which a fulfill-reaction is registered immediately. Note that entire expression e_0 may form the beginning of a longer chain of promises. In such cases, it is often possible to shorten the length of the promise-chain by refactoring e_0, e.g., to `Promise.resolve(f(`$\texttt{e}_1$`)}`, to reduce the number of created promises. Section 4.3 discussed a slightly more complex instance of this anti-pattern.

executorOneArgUsed. This anti-pattern targets expressions of the form `new Promise(`f`)` where a promise is constructed using an executor function f that has formal parameters v_0 and v_1 (usually the parameters of executor functions are called `resolve` and `reject` but programmers may choose different names). Furthermore, an additional constraint is imposed that if the body of f contains expressions e_1 and e_2 that refer to v_0 or v_1, then they must both refer to the same variable. In other words, the anti-pattern targets executor functions that either resolve or reject the promise, but not both. In such cases, it may be possible to refactor the code to use `Promise.resolve` or `Promise.reject` instead.

reactionReturnsPromise. In this scenario, a reaction f that is registered on a promise in an expression of the form e_1`.then(`f`)` returns an expression e_2 that consists of either a call to `Promise.resolve` or a call to `Promise.reject`. In such cases, it is often possible to avoid

the explicit construction of a promise because the reaction already creates a promise that is fulfilled or rejected with the reaction's return value.

customPromisification. This anti-pattern aims to detect situations where a programmer has written a custom function for promisifying an event-based API call. It targets expressions of the form `new Promise`(f_0) where the Promise constructor is invoked with an executor function that contains a call $f_1(..., f_2)$, that passes a callback function f_2 to some API function f_1. Moreover, f_2 contains a statement `if` $(\cdots)$ $\{\mathtt{s}_0\}$ `else` $\{\mathtt{s}_1\}$, where either s_1 calls the function passed as the first parameter to the executor (usually called `resolve`) and s_2 calls the function passed as the second parameter to the executor (usually called `reject`), or vice versa. In such cases, it is often possible to utilize the `util.promisify` promisification function instead. While this does not reduce the number of promises created, it avoids the pitfalls of accidentally introducing bugs when re-implementing functionality that is available in standard libraries.

explicitPromiseConstructor. This anti-pattern occurs when a new promise is constructed that is fulfilled when some existing promise is fulfilled, and that is rejected when that promise is rejected. Concretely, we say that an instance of this pattern occurs when the promise constructor is invoked with an executor function f_0 that has parameters v_0 and v_1. In addition, the body of f_0 contains an expression e_1.`then`(f_1).`catch`(f_2), where f_1 has a parameter v_2 and f_2 has a parameter v_3. Lastly, f_1 is required to contain a call $v_0(v_2)$ and f_2 is required to contain a call $v_1(v_3)$. Occurrences of this anti-pattern can often be refactored to avoid the creation of a new promise, e.g., by returning the promise e_1.

4.5 Implementation

DrAsync consists of three components: (i) a static analysis for detecting anti-patterns, (ii) a dynamic analysis for gathering information about the lifetimes of promises and detecting run-time instances of anti-patterns, and (iii) an interactive profiling tool that visualizes the lifetimes of promises and instances of anti-patterns, and that provides additional features for understanding execution behavior. Our code is open-source and publicly available [5].

[5] Artifact link: https://doi.org/10.5281/zenodo.5915257

4.5.1 Static Analysis

The static analysis uses CodeQL [51, 5] to implement the anti-patterns of Figure 4.3 as a set of QL queries. These queries follow the logic of the definition closely. For example, the query that is used to find the *promiseResolveThen* anti-pattern looks as follows:

```
107  predicate promiseDotResolveDotThen(MethodCallExpr c) {
108    c.getMethodName() = "then" and
109    c.getReceiver() instanceof MethodCallExpr and
110    ((MethodCallExpr) c.getReceiver()).getMethodName() = "resolve"
111  }
```

In two cases, we extended the queries with special handling of corner cases. Our implementation of *executorOneArgUsed* was extended to exclude cases where calls to `resolve` are passed as an argument to `setTimeout` as we found that such occurrences of the anti-pattern are generally not amenable to refactoring. We also extended *loopOverArrayWithAwait* to handle `for-of` and `for-in` loops.

4.5.2 Dynamic Analysis

DrAsync relies on the Node.js Async hooks API [3] to instrument source code to log the creation and settlement of promises, to record when `await`-expressions are first encountered and when their execution is resumed, and to determine run-time instances of anti-patterns. The instrumentation distinguishes different run-time instances of promises that are created at the same location (e.g., promises created during multiple executions of the same promise constructor or of the same `async` function), enabling us to calculate how often each anti-pattern is executed.

Furthermore, information is recorded about dependencies between promises: the Async hooks API provides a unique `asyncId` for each promise, as well as a `triggerAsyncId`, which is the `asyncId` of the promise that triggered it (i.e., the promise that it depends on). Moreover, the dynamic analysis determines whether promises are related to I/O operations through simple heuristics (if a promise originates from a function from a predefined list of I/O functions from the `util` Node.js library), and whether they originate from user code or from library code. This information is used in the interactive visualization to enable programmers to filter promises based on their origin, and quickly hone-in on relevant promises.

The results of the static analysis and a dynamic analysis are aggregated into a single trace file that is used in *DrAsync*'s interactive visualization component.

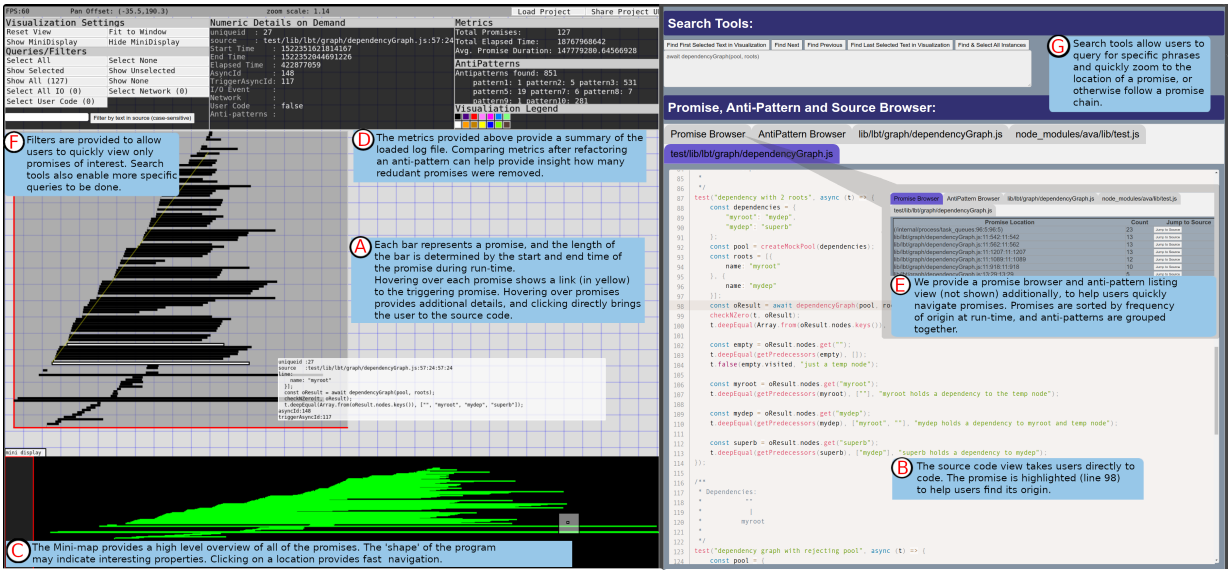


Figure 4.4: The interactive visualization displays the run-times of each promise as well as visually summarizes the data capture by DrAsync. Users can filter particular promises and directly investigate the source code for more details on demand.

4.5.3 Interactive Visualization

The visualization helps with exploring the execution behavior of asynchronous JavaScript code and enables one to identify certain anti-patterns visually. The visualization also shows the number of runtime occurrences for each instance of an anti-pattern, enabling programmers to prioritize those anti-pattern instances that may impact execution behavior the most.

DrAsync's interactive visualization tool was developed using the P5.js framework [148]. Figure 4.4 shows a screenshot of a visualization produced by *DrAsync*, which follows the standard information taxonomy by providing: a high level overview, filters, and details on demand [197]. We briefly discuss *DrAsync*'s different views.

Promise Lifetime View and Source Code View. This view (labeled Ⓐ in the figure) is organized as a Gantt Chart [106]. Here the x-axis represents time, and the y-axis shows the created promises as a series of stacked bars, so each promise is represented by one line that starts at the time when the promise was created, and that ends when it was settled. Users can pan and zoom through the promise lifetime view, and hovering on a promise shows a fragment of the source code responsible for creating the promise, along with some

Table 4.1: Summary of Case Study

Anti-Pattern	# Successful	# Unsuccessful
asyncFunctionNoAwait	9	1
asyncFunctionAwaitedReturn	9	1
loopOverArrayWithAwait	7	3
promiseResolveThen	9	1
executorOneArgUsed	6	4
reactionReturnsPromise	9	1
customPromisification	9	1
explicitPromiseConstructor	7	3

meta-information. Furthermore, clicking on one of the promises opens the associated source code in tab Ⓑ for further inspection.

Mini Display View. This view (green bars in the view labeled Ⓒ at the bottom of the figure) shows the general 'shape' of the promises created during execution; clicking here enables the user to quickly navigate to areas of interest in the promise lifetime view (e.g., staircase patterns corresponding to instances of *loopOverArrayWithAwait* that may benefit from refactoring).

Metrics View. This view, labeled Ⓓ, summarizes metrics: how many promises were created, the total elapsed time, the average duration of promises, and counts for detected anti-patterns. These can be compared before and after refactoring to see if redundant promises have been eliminated, or if performance has changed.

Summary View and Filters. This view, labeled Ⓔ, shows of all promises and anti-pattern instances; clicking on these will navigate to the associated promise in the promise lifetime view, and will display the associated source code. For realistic applications, the number of promises created at run-time can quickly become overwhelming, so *DrAsync* provides various filtering facilities to focus on promises of interest. In particular, users can focus on those promises that are related to file I/O or network I/O (see view labeled Ⓕ), or on promises whose creation site matches a specified text string (see view labeled Ⓖ).

4.6 Case Study

To evaluate if the anti-patterns reported by *DrAsync* represent useful information, we randomly selected 10 instances of each anti-pattern and attempted to refactor them manually.

These 10 instances were chosen from the 20 subject applications that we will report further on in Section 4.7. To ensure that our findings are not biased towards a particular programming style, no more than three instances of each pattern were chosen from a single application, and we only selected anti-pattern instances that *DrAsync* reported as being executed by the application's test suite, so that we could check that the refactoring did not cause behavioral changes.

An overview of our findings can be found in Table 4.1. Below we report on some noteworthy situations that we encountered. Many refactorings were simple and quick, though others took more considerable time (e.g., some loop refactorings took ¿15 minutes in order to understand possible data dependencies). Further details for all 80 cases can be found in Appendix A.

asyncFunctionAwaitedReturn. As discussed in Section 4.4, this anti-pattern reflects inefficient code as it involves waiting for a promise to settle with some value v, and then creating a new promise that is settled with the same value. The following function in file `/src/utils/readSpec.ts` in *openapi-typescript-codegen* was flagged by *DrAsync* as an instance of this anti-pattern:

```
112  export async function readSpec(input: string): Promise<string> {
113    if (input.startsWith('https://')) {
114      return /*await*/ readSpecFromHttps(input);
115    }
116    if (input.startsWith('http://')) {
117      return await readSpecFromHttp(input); // not executed
118    }
119    return /*await*/ readSpecFromDisk(input);
120  }
```

Here, `await` is redundantly used on each of the return paths and *DrAsync* informed us that the first and third of these await-expressions were executed by the test suite. We confirmed that the tests still passed after removing the `await` keywords.

loopOverArrayWithAwait. Section 4.3 already discussed an instance of this anti-pattern in *appcenter-cli* that we were able to refactor successfully. However, some of the instances reported by *DrAsync* could not be refactored, such as the the following code snippet on lines 159–162 in file `/src/TemplateLayout.js` in *eleventy*:

```
121  for (let fn of fns) {
122    templateContent = await fn(data);
123    data = TemplateLayout
124             .augmentDataWithContent(data, templateContent);
125  }
```

Here, each loop iteration awaits the result of the call to `fn(data)` and then re-assigns `data` on the next line. Since each loop iteration depends on a value computed in the previous iteration, we are unable to parallellize the loop using `Promise.all`.

executorOneArgUsed. An interesting case of this anti-pattern occurs on lines 39-56 in `src/streaming/utils/CapabilitiesFilter.js` in *dash.js*:

```
126 return new Promise((resolve) => {
127   const promises = // details omitted
128   Promise.all(promises)
129     .then(() => { /* details omitted */ resolve(); })
130     .catch(() => { resolve(); });
131 });
```

Here, a new promise is created that is fulfilled (with the value `undefined` since no argument is passed to `resolve`) in reactions on a promise that is created by a call to `Promise.all`. The creation of a new promise can be avoided by refactoring the above code to:

```
132 const promises = // details omitted
133 return Promise.all(promises)
134     .then(() => { /* details omitted */ return; })
135     .catch(() => { return; });
```

After this refactoring, it is evident that the resulting code lacks proper error handling, given that `catch` is used to register a no-op function to "absorb" errors that cause the previous reaction in the promise chain to be rejected.

customPromisification For this anti-pattern, we found that we could successfully refactor 9 of 10 instances highlighted by the tool using the `util.promisify` library function. The remaining case involved the use of an event handler with complex control flow.

In all but one of the successful cases, using `promisify` and refactoring the inner logic of the callback into a reaction on a call to the promisified function was sufficient. For a more interesting case, consider:

```
136 return async function (data) {
137   /* return new Promise(function (resolve, reject) {
138     tmpl.render(data, function (err, res) {
139       if (err) {
140         reject(err);
141       } else {
142         resolve(res);
143       }
144     });
145   }); */
146   const tmplRenderProm = util.promisify(tmpl.render);
147   return tmplRenderProm.call(tmpl, data);
148 };
```

This snippet is from lines 467-475 in `eleventy`'s file `src/Engines/Nunjucks.js`. Here, `tmplRenderProm` must be invoked with `Function.prototype. call` to preserve the correct value for `this` during its execution.

reactionReturnsPromise For this anti-pattern, 9 of the 10 cases we examined could be refactored; the one unsuccessful case involved a promise reaction with complex event-handlers, where the returned promise was fulfilled or rejected in response to external events.

For an example of a successful refactoring, consider this snippet from `netlify-cms`, lines 428-433 of `packages/netlify-cms- core/src/backend.ts`:

```
149  const publishedEntry = await this.implementation
150    .getEntry(path)
151    .then(({ data }) => data)
152    .catch(() => {
153      // return Promise.resolve(false);
154      return false;
155    });
```

Here, `.catch` and `.then` return promises anyway, so explicitly returning a promise that is immediately fulfilled or rejected is needless.

4.7 Evaluation

This evaluation aims to answer the following research questions:

RQ1: How often do the anti-patterns of Figure 4.3 occur in practice?

RQ2: How often can anti-patterns reported by *DrAsync* be eliminated using refactoring?

RQ3: Can the elimination of anti-patterns improve performance?

RQ4: What is the performance of *DrAsync*?

4.7.1 Experimental Setup

To identify a set of candidate projects, we first ran a CodeQL query (on a large set of JavaScript GitHub repositories available to the CodeQL team) to find projects containing

Table 4.2: Subject Applications

Project *(links to repos @ SHA)*	**SHA**	**KLOC**	**Anti-Patterns** #	**Anti-Patterns** / KLOC	**# Files**	**# Funs**	**Tests**
appcenter-cli	2109d1	96	73	0.76	2645	8406	434
Boostnote	58c4a7	32	29	0.92	276	4572	81
browsertime	648e16	223	134	0.60	197	17557	13
CodeceptJS	68ad16	19	398	21.5	180	3583	34
dash.js	996e21	20	70	3.5	123	3598	18
eleventy	6776e8	53	65	1.2	358	5532	1070
erpjs	5ddcb7	30	139	4.6	295	4509	973
fastify	aee28e	136	2	0.01	108	20461	54
flowcrypt-browser	bc0d348	41	296	7.1	240	7119	5394
media-stream-library-js	4dd02a	37	184	5.0	117	4754	154
mercurius	97ee14	60	22	0.37	220	4969	959
netlify-cms	071b05	12	77	6.6	118	4009	73
openapi-typescript-codegen	715ddc	34	9	0.27	180	4529	1092
rmrk-tools	64c8cf	36	334	9.2	301	7916	247
stencil	0c2e95	193	265	1.4	326	45025	1619
strapi	1fe4b5e	80	198	2.5	292	4875	982
treeherder	b70d3b	37	50	1.4	154	4004	300
ui5-builder	7490fb	44	77	1.8	216	4802	741
vscode-js-debug	2af8cb	78	150	1.9	300	11496	186
vuepress	f077f7	14	19	1.3	276	7736	104

Table 4.3: Run Times

Project	**QLDB Build Time (s)**	**Test Time (Before/After)** Mean	StDev	Mean	StDev	**Overhead of Instrumentation**
appcenter-cli	126.172	31.45	1.05	34.29	0.86	9.03%
Boostnote	40.069	41.50	0.80	43.59	2.23	5.03%
browsertime	29.61	0.55	0.01	0.66	0.01	20.59%
CodeceptJS	57.448	2.83	0.02	3.16	0.02	11.62%
dash.js	59.681	4.12	0.16	5.77	0.26	39.79%
eleventy	34.446	21.62	0.27	50.93	0.36	135.6%
erpjs	106.687	19.0	0.23	21.15	0.27	11.37%
fastify	42.472	118.58	0.67	127.43	1.02	7.47%
flowcrypt-browser	1064.285	1.77	0.03	2.27	0.05	1.28%
media-stream-library-js	63.543	122.88	0.85	131.52	1.36	7.03%
mercurius	42.099	55.17	0.51	65.44	0.65	18.62%
netlify-cms	94.35	504.42	2.30	605.48	1.69	20.04%
openapi-typescript-codegen	45.618	45.56	0.62	56.10	0.57	23.04%
rmrk-tools	326.839	38.01	0.32	41.42	0.39	8.97%
stencil	823.68	453.33	1.67	484.40	5.89	6.85%
strapi	77.734	164.89	0.95	195.13	2.06	18.33%
treeherder	43.12	209.79	1.06	229.93	2.60	9.60%
ui5-builder	44.462	31.82	0.23	69.14	0.49	117.31%
vscode-js-debug	127.798	1.39	0.02	2.31	0.06	65.48%
vuepress	81.301	6.97	0.20	22.64	0.96	224.79%

Table 4.4: Anti-pattern stats. Legend: P1 = *asyncFunctionNoAwait*, P2 = *loopOverArrayWithAwait*, P3 = *asyncFunctionAwaitedReturn*, P4 = *explicitPromiseConstructor*. "S" stands for *static occurrences*; "E" stands for *static occurrences that are dynamically executed*; "D" stands for *the total number of runtime promises associated with this anti-pattern.*

Project	P1		P2		P3		P4	
	S (E)	D	S (E)	D	S (E)	D	S (E)	D
appcenter-cli	23 (1)	42	11 (0)	0	18 (0)	0	1 (0)	0
Boostnote	1 (0)	0	0 (0)	0	0 (0)	0	3 (3)	6
browsertime	105 (1)	3	21 (1)	47	0 (0)	0	1 (0)	0
CodeceptJS	357 (3)	39	33 (0)	0	1 (0)	0	0 (0)	0
dash.js	0 (0)	0	0 (0)	0	0 (0)	0	23 (8)	224
eleventy	39 (24)	4416	10 (10)	884	9 (7)	1271	0 (0)	0
erpjs	40 (0)	0	12 (0)	0	66 (1)	36	0 (0)	0
fastify	0 (0)	0	0 (0)	0	0 (0)	0	0 (0)	0
flowcrypt-browser	79 (0)	0	50 (0)	0	150 (0)	0	2 (0)	0
media-stream-library-js	56 (0)	0	3 (0)	0	121 (1)	1	0 (0)	0
mercurius	14 (3)	72	4 (3)	322	0 (0)	0	0 (0)	0
netlify-cms	45 (3)	1261	8 (0)	0	5 (0)	0	0 (0)	0
openapi-typescript-codegen	2 (1)	2	3 (0)	0	2 (2)	28	0 (0)	0
rmrk-tools	241 (0)	0	43 (0)	0	18 (0)	0	0 (0)	0
stencil	123 (1)	74	33 (3)	217	20 (2)	35	1 (0)	0
strapi	81 (5)	179	45 (6)	100	26 (0)	0	4 (0)	0
treeherder	43 (7)	211	2 (2)	10	0 (0)	0	0 (0)	0
ui5-builder	51 (25)	1510	5 (3)	373	1 (1)	23	2 (2)	69
vscode-js-debug	94 (2)	84	7 (0)	0	20 (3)	749	1 (0)	0
vuepress	7 (0)	0	3 (2)	3448	1 (1)	1910	1 (0)	0
Summary	1401 (76)	7893	293 (30)	5401	458 (18)	4053	39 (13)	299

promise-related features[6]. Of the ¿100K projects that this turned up, we used the *npm-filter* [50] tool to discard projects that did not have running test suites, resulting in 450 projects with at least one running test command. Of those projects, we randomly selected 20 projects meeting the following criteria: the project (i) was edited in the last year, (ii) had over 20 stars, (iii) contained over 20 instances of promise-related features, and (iv) running the application's test suite results in the creation of at least 40 promises.

All experiments were performed on a CentOS Linux 7.8.2003 (Core) server, with 2x 32-core 2.35GHz processors, and 128GB RAM.

4.7.2 RQ1: How often do anti-patterns occur?

After discounting anti-patterns occurring in test code, compiled TypeScript, and distributions, we found 2.6k anti-patterns instances in the 20 projects selected for evaluation.

[6]This includes: references to the `Promise` constructor, references to `Promise.resolve`, `Promise.reject`, `Promise.all`, `Promise.race`, and `Promise.any`, references to methods with names `then` or `catch`, `async` functions, and `await` expressions.

Table 4.5: Anti-pattern stats. Legend: P4 = *explicitPromiseConstructor*, P5 = *customPromisification*, P6 = *promiseResolveThen*, P7 = *reactionReturnsPromise*, P8 = *executorOneArgUsed*. "S" stands for *static occurrences*; "E" stands for *static occurrences that are dynamically executed*; "D" stands for *the total number of runtime promises associated with this anti-pattern.*

Project	**P5**		**P6**		**P7**		**P8**	
	S (E)	D	S (E)	D	S (E)	D	S (E)	D
appcenter-cli	14 (3)	446	1 (0)	0	4 (1)	4	1 (0)	0
Boostnote	9 (5)	18	5 (2)	7	5 (0)	0	6 (1)	1
browsertime	1 (0)	0	2 (0)	0	0 (0)	0	4 (0)	0
CodeceptJS	1 (0)	0	3 (3)	1125	0 (0)	0	3 (0)	0
dash.js	2 (2)	55	0 (0)	0	27 (0)	0	18 (10)	188
eleventy	1 (1)	244	0 (0)	0	5 (4)	31	1 (1)	6
erpjs	14 (0)	0	6 (0)	0	0 (0)	0	1 (0)	0
fastify	0 (0)	0	0 (0)	0	0 (0)	0	2 (2)	25
flowcrypt-browser	3 (0)	0	0 (0)	0	0 (0)	0	12 (0)	0
media-stream-library-js	2 (0)	0	0 (0)	0	0 (0)	0	2 (1)	2
mercurius	3 (3)	409	1 (1)	10	0 (0)	0	0 (0)	0
netlify-cms	0 (0)	0	4 (1)	10	10 (2)	14	5 (1)	2286
openapi-typescript-codegen	1 (0)	0	0 (0)	0	0 (0)	0	1 (1)	4
rmrk-tools	8 (0)	0	2 (0)	0	0 (0)	0	22 (0)	0
stencil	17 (1)	3	21 (0)	0	1 (0)	0	49 (0)	0
strapi	19 (0)	0	8 (1)	20	12 (5)	5	3 (0)	0
treeherder	2 (0)	0	0 (0)	0	3 (3)	61	0 (0)	0
ui5-builder	5 (2)	56	5 (5)	896	2 (2)	310	6 (2)	50
vscode-js-debug	4 (0)	0	0 (0)	0	2 (0)	0	22 (2)	42
vuepress	0 (0)	0	5 (0)	0	1 (0)	0	1 (0)	0
Summary	106 (17)	1231	63 (13)	2068	72 (17)	425	159 (21)	2604

Moreover, *DrAsync*'s dynamic analysis detected that a total of 24K instances of these anti-patterns were executed by the applications' test suites. These results are tabulated in Table 4.4, and provide strong evidence that anti-patterns commonly occur. The first cells of the table read: `appcenter-cli` has 23 instances of the *asyncFunctionNoAwait* pattern in its code (S), 1 instance is executed in the tests (E), and 42 runtime promises are associated with this anti-pattern (D).

Anti-patterns commonly occur in asynchronous JavaScript code. We found a total of 2.6K anti-patterns in 20 subject applications.

4.7.3 RQ2: Can detected anti-patterns be refactored?

Section 4.6 summarized findings of a case study wherein we tried to refactor 80 instances of anti-patterns flagged by *DrAsync*. Of these 80 cases, we were able to successfully refactor 65. For the 15 that we were unable to refactor, not all are necessarily false positive, because developers with more expert knowledge may have additional insights enabling them to refactor the code. Each of the refactorings is reported on in Appendix A.

A case study involving 80 anti-patterns in real-world code suggests that the majority of anti-patterns detected by *DrAsync* can be eliminated through refactoring.

4.7.4 RQ3: Can the elimination of anti-patterns improve performance?

Generally speaking, we would expect the elimination of an anti-pattern to impact performance only in significant ways if the anti-pattern is executed many times, if the refactoring results in the elimination of a large number of promises at run-time, or if the refactoring enables additional concurrency. We examined three refactorings in our case study that meet some of these criteria, for which we crafted experiments that emphasize the performance of the code fragment in question.

appcenter-cli/`cpDir`. This particular instance of the *loopOverArrayWithAwait* anti-pattern was previously discussed in Section 4.3 and involves a function that copies one directory to another. We chose this anti-pattern instance as the correctness of the refactoring was easy to confirm, and we could easily craft a controlled experiment; in this experiment,

we executed `cpDir` 50 times on a large directory of 7.8G with 37 files, and found that the refactored version ran 16.4% (4.8s vs 5.8s) faster on average than the original, and that the variance between run times was 37.9% smaller (0.33s vs 0.54s), leading to more predictable performance.

***vuepress*/`apply`.** This function contains a loop exhibiting the *loopOverArrayWithAwait* anti-pattern:

```
156  for (const { value, name: pluginName } of this.appliedItems) {
157    // details omitted
158    await ctx.writeTemp(`${dirname}/${name}`, ... );
```

We chose to focus on this anti-pattern instance because the correctness of the refactoring was easy to check, and the code is frequently invoked by the test suite, so we can observe performance in a realistic use-case. After refactoring this code fragment to use `Promise.all`, we ran the application's test suite 50 times on the versions before and after the refactoring. The results show that the refactoring reduced the time needed to execute this code fragment by 36.1% on average, and that run time variability was reduced by 16%.

***strapi*/`evaluate`.** This instance of the *promiseResolveThen* anti-pattern occurs in the `strapi` application:

```
159  // const evaluatedConditions = await Promise.resolve(conditions)
160  //     .then(resolveConditions)
161  //     .then(filterValidConditions)
162  //     .then(evaluateConditions)
163  //     .then(filterValidResults);
164  const evaluatedConditions = filterValidResults(await
165    evaluateConditions(filterValidConditions(
166    resolveConditions(conditions))));
```

We selected an instance of this anti-pattern to assess the performance impact of eliminating more than just the *loopOverArrayWithAwait* anti-pattern, and we selected this instance specifically as it is frequently executed by the test suite and involves many chained promises (our refactoring eliminates 5 runtime promises per execution of this snippet). We refactored this fragment to instead call the functions directly (the code exhibiting the anti-pattern is commented). We ran the `strapi` test suite 50 times and observed that the refactoring reduced the average time needed to execute this code fragment by 4%, and the standard deviation by 7.4%.

Full Test Suite Refactorings We refactored every executed instance of an anti-pattern in the `eleventy` project, and timed the execution of the test suite before and after. We

found that roughly 1.1k fewer user promises (39,978 to 38,748) were created, and found no meaningful change in the run time of the test suite. We performed a similar case study with `vuepress`. We again found no meaningful change in test suite execution time, and found roughly 1.2k fewer user promises (32,264 to 31,021).

Note that we chose these projects to fully refactor as they had a few anti-patterns that had many associated dynamic promises, and the refactorings were simple enough such that we could verify their correctness.

Discussion Overall, it is difficult to measure the effect of the removal of runtime promises on the overall performance of applications, due mostly to their asynchronous nature. Even if thousands of redundant promises are eliminated, it is possible that the application was waiting on another operation which takes longer than the sum total of the lifetimes of the eliminated promises.

The elimination of anti-patterns reduces the number of promises created and enables additional parallelism, which may speed up the execution of the affected code fragments.

4.7.5 RQ4: What is the performance of *DrAsync*?

There are three main components to the run time of *DrAsync*.

First, the time to build the QL databases is reported in column **"QLDB Build Time"** in Table 4.2—the build times vary, but are only exceptionally high for `flowcrypt-browser` and `rmrk-tools`. Note that this only needs to be done once per project (it needs to be rebuilt when code changes, however), and the database can be reused for other CodeQL queries; linting, by comparison, would be much faster but cannot detect all of the anti-patterns detected by *DrAsync*. To put this number into context, the mean run time of the test suites are found under the first **Mean** column.

Second, the time to run the anti-pattern detection queries is quite low: we ran 160 queries (8 anti-patterns $\times$ 20 projects) in sequence, and only 14 of the 160 query/project combinations took over 30s, and the mean run time was 18.4s. The full query run times are available in Appendix A.

Finally, *DrAsync*'s dynamic analysis adds roughly 27% performance overhead (harmonic mean from column **Overhead of Instrumentation**). Note that, for the **Mean** columns under **Test Time (Before/After)**, the means reported are taken over 20 test suite executions, and the standard deviation of those runs is reported in the **StDev**

columns. The overhead was calculated by dividing the mean test suite execution time with instrumentation by the mean test suite execution time without instrumentation. Importantly, note that the subject applications vary wildly in size, and *DrAsync*'s run time is reasonable in all cases.

DrAsync runs quickly, and the performance of the tool scales well as code size increases.

4.8 Threats to Validity

There are several factors that threaten the validity of our results. First, the selection of subject applications used for our evaluation may not be representative. We attempted to mitigate this by randomly selecting applications that met specified criteria that made them suitable subjects for analysis. Also, note that the subject applications include popular and well-maintained projects from major vendors such as Microsoft and SAP. Second, the anti-pattern instances selected in our case study may not be representative. We attempted to mitigate this by randomly selecting these instances, and selected no more than three instances from any one project. Third, our experiences in manually refactoring the anti-pattern instances may be subject to bias and errors. To mitigate the risk of mistakes in the manual refactorings, we focused on anti-pattern instances that are executed by the application's test suite so that we could check for behavioral differences by running the tests. As for bias, we were unfamiliar with the source code for the subject applications, we made an effort to randomly select subjects for the case study, and we highlighted both positive and negative refactoring experiences. Finally, regarding the performance implications of eliminating anti-patterns, one may object that the observed speedups are small and only apply to code fragments in three selected subject applications, under idealized conditions. This is correct, and we do not make broader claims in this regard.

4.9 Relation to Previous Research

Chapter 3 touches on general background for this work, namely static and dynamic analysis of JavaScript, program understanding, and refactoring. There are some branches of the literature that relate specifically to this chapter, namely: detection of anti-patterns in JavaScript software, profiling concurrent applications, and performance visualization. These were not discussed as part of the general related work in Chapter 3, and will be outlined here.

4.9.1 JavaScript Anti-Patterns

The detection and remediation of anti-patterns in software has long been a part of good software development practices. Chapter 3 in Fowler's seminal book on refactoring [92] enumerates a number of "code smells" that can be addressed using the refactorings presented in the later chapters.

Several tools for static analysis and style have been developed [20, 2, 12] that check a broad range of rules for identifying potential quality issues in JavaScript software. ESLint [12] supports several rules concerned with `async/await` such as `no-await-in-loop` for detecting the use of `await` in loops. Our research goes beyond ESLint by considering a broader range of asynchronous anti-patterns, visualizing the behavior of asynchronous applications, and combining more sophisticated static analysis and dynamic analysis. Further, ESLint only detects three of the eight anti-patterns reported in this chapter: *loopOverArrayWithAwait*, *asyncFunctionAwaitedReturn*, and *asyncFunctionNoAwait* (ESLint flags *any* loop with an await inside, while our anti-pattern is specific to loops over arrays, which in our experience is more likely to amenable to refactoring). ESLint also currently does not support the data-flow analysis required to detect several anti-patterns described in the chapter.

Madsen et al. [142] defined the *event-based call graph*, which extends the traditional notion of a call graph with nodes and edges that reflect the flow of control due to event-handling. Recently, Arteca et al. [48] presented a statistical analysis for detecting event listeners that are likely to be dead code due to bugs in event-handling code.

Madsen et al. [140] presented a formal semantics for JavaScript promises, and defined the *promise graph* capturing relationships between promises, and use it to identify bugs found on StackOverflow. Alimadadi et al. [43] present PromiseKeeper, a tool that constructs promise graphs using dynamic analysis, defining a number of dynamic anti-patterns in promise graphs such as unhandled promise rejections. The work by Madsen et al. and Alimadadi et al. predates JavaScript's `async/await` feature. While our work and PromiseKeeper are concerned with the visualization of execution behavior of promise-based code, the visualizations are very different: PromiseKeeper provides a fine-grained visualization of promises and the functions and values they interact with, whereas our work is focused on a large-scale visualization that is focused on the performance aspects of promises and await-expressions.

The academic community has also focused on the detection of code smells in JavaScript code that are unrelated to asynchrony. Nguyen et al. [166] present a tool for detecting embedded code smells in web applications using dynamic analysis. Fard and Mesbah [88] identify 13 code smells that commonly arise in JavaScript software and present a technique

based on static and dynamic analysis to detect them. Johannes et al. [118] report on a large-scale empirical study that investigates the relation between code smells in JavaScript software and the fault-proneness of the program parts containing the code smells. Gong et al. [101] present DLint, a tool for detecting code quality issues using dynamic analysis rather than the traditional static analysis.

4.9.2 Profiling Concurrent Applications

Early work in this area by Waheed and Rover [226] considered techniques for visualizing the performance of parallel programs at the processor level, using techniques from the scientific visualization community. Miller et al. [156] present Paradyn, a tool for measuring and visualizing the performance of large-scale parallel programs using an adaptive instrumentation targeted at long-running applications. Paradyn differs from our work in that it selectively instruments code and visualizes the program as a graph using a graph coloring technique. Meira et al. [119] present Carnival, a performance measurement tool for determining the underlying causes for waiting time in distributed memory systems, again at the processor level. Carnival differs in that it measuring wait times that rely on synchronization primitives used on multi-processor (as opposed to single core) systems.

Joao et al. [117] present a technique for detecting performance bottlenecks in multi-threaded applications (critical sections, barriers, and slow pipeline stages) that have the effect of serializing program execution. Unlike [117], our technique is implemented entirely using source code instrumentation and our focus is on visualizing anti-patterns so that users can remedy them manually.

Dutta et al. [72] present a technique for classifying performance bottlenecks in multi-threaded applications, differentiating between *on-chip* and *off-chip* Unlike our approach, Dutta's only provides an overall assessment, and it does not identify specific regions in the code that constitute the most significant performance bottlenecks.

4.9.3 Software Visualization

Recent work by Tominaga et al. [213] built a tool called AwaitViz to capture instances of async/await in order to visualize execution order focus on improving programmer comprehension of the code. Additional visualizations on understanding async/await was done by Sun et al. by generating Async Graphs [205]. The async graphs are used to help identify bugs related to asynchronous execution and primarily focus on *when* specific events

happen during the asynchronous flow of execution in Node.js applications for bug detection. Additional concurrency profiling tools with visualizations in IDEs have been created, focusing on multi-threaded applications and resource utilization: JetBrains's PyCharm Thread Concurrency Visualization [25], Visual Studio's Concurrency Visualizer [1], and Intel's VTune [26].

4.10 Conclusion

We identified 8 anti-patterns that commonly occur in JavaScript code that uses promises and `async`/`await`. We presented *DrAsync*, a tool that relies on a combination of static and dynamic analysis to detect instances of anti-patterns, and that provides an interactive visualization to help programmers quickly diagnose quality issues and performance bottlenecks in their asynchronous applications.

In an empirical evaluation, *DrAsync* detected 2.6K anti-patterns in 20 subject applications, which were executed 24K times in the aggregate. We report on a case study in which we manually attempted to refactor 10 instances of each anti-pattern, concluding that the majority of *DrAsync*'s findings are actionable, and that refactoring anti-patterns may improve the performance of the affected code.

4.11 Discussion

In this chapter, we proposed a diagnostic tool for detecting misuses of promise-related features in JavaScript. In a sense, the static analysis anti-pattern detection queries can be thought of as "super linters", in that that they identify code issues (like linters), but use more sophisticated static analysis techniques like data flow analysis (unlike linters). Broadly, it appears worth investigating how linters can be enhanced by incorporating more precise static analysis techniques; in this work, a little data flow went a long way.

In Chapters 5 and 7, static analyses not only identified issues in the code, but also generated information about the code fragments that were precise enough to be leveraged to automatically suggest code transformations to fix issues. In contrast, the anti-patterns detected by *DrAsync* are not as straightforward to fix; in Section 4.6, we found no general formula that could be applied to fix every instance of any given anti-pattern. Instead, programmers are invited to further investigate the highlighted issues, which is made easier thanks to the visualization and profile information, and we showed that many anti-patterns

could be fixed by complete outsiders to the code bases, suggesting that the insights gleaned from *DrAsync* are indeed actionable. One interesting avenue of future work in this space would be to leverage the precise *dynamic* information from execution profiles to help construct automated code transformations.

4.12 Data Availability

Experimental data associated with this research is available on Zenodo: https://doi.org/-10.5281/zenodo.5428997. A software artifact is also available on Zenodo: https://doi.org/-10.5281/zenodo.5915257.

Chapter 5

Database Usage Optimizations

Abstract

An Object-Relational Mapping (ORM) provides an object-oriented interface to a database and facilitates the development of database-backed applications. In an ORM, programmers do not need to write queries in a separate query language such as SQL, they instead write ordinary method calls that are mapped by the ORM to database queries. This added layer of abstraction hides the significant performance cost of database operations, and misuse of ORMs can lead to far more queries being generated than necessary. Of particular concern is the infamous "N+1 problem", where an initial query yields N results that are used to issue N subsequent queries. This anti-pattern is prevalent in applications that use ORMs, as it is natural to iterate over collections in object-oriented languages. However, iterating over data that originates from a database and calling an ORM method in each iteration may result in suboptimal performance. In such cases, it is often possible to reduce the number of round-trips to the database by issuing a single, larger query that fetches all desired results at once.

We propose an approach for automatically refactoring applications that use ORMs to eliminate instances of the "N+1 problem", which relies on static analysis to detect data flow between ORM API calls. We implement this approach in a tool called REFORMULATOR, targeting the Sequelize ORM in JavaScript, and evaluate it on 8 JavaScript projects. We found 44 N+1 query pairs in these projects, and REFORMULATOR refactored all of them successfully, resulting in improved performance (up to 7.67x) while preserving program behavior. Further experiments demonstrate that the relative performance improvements grew as the database size was increased (up to 38.58x), and that front-end page load times were improved.

5.1 Introduction

An ORM (Object-Relational Mapping) provides an object-oriented facade for a database enabling programmers to access it using ordinary method calls. The ORM maps such method calls to database queries and converts query results to objects in the host language so that programmers do not need to use a separate database query language like SQL to interact with the database. However, the added layer of abstraction introduced by ORMs may obscure the cost of database operations, and careless ORM usage may generate more database queries than are necessary, causing poor performance.

Of particular concern is the infamous "N+1 problem" [61, 240, 58], which arises when an initial database query yields N results that are then used to issue N subsequent database queries. This can lead to significant performance problems because database queries are typically high-latency operations. The "N+1 problem" anti-pattern frequently occurs in applications that use ORMs, where it often arises in the following scenario:

- An initial call to the ORM's Application Programming Interface (API) generates a database query that results in a collection C of objects.
- Then, a loop iterates through C and, for each element $c \in C$, calls an ORM API method with c as an argument, resulting in the generation of another new database query.

We found that, in many of these cases, the "N+1 problem" can be remediated by inserting a single ORM API call that has the effect of retrieving the information from the database that was previously fetched by the N subsequent queries. This refactoring, by significantly reducing the number of round-trips to the database, can drastically improve performance.

We present an approach for automatically detecting instances of the "N+1 problem" and generating code transformations that reduce the number of database queries. To detect instances of the "N+1 problem", a static data-flow analysis detects data flow from the result of one ORM API call to an argument of another ORM API call, where the latter call occurs within a loop. To repair these instances, we define a set of declarative rewrite rules that specify how code should be transformed to reduce the number of generated queries. These transformations result in code that: (i) issues a constant number of queries, (ii) is behaviorally equivalent, and, importantly, (iii) performs better and scales as database size increases.

We implement this technique in a tool called REFORMULATOR, targeting the Sequelize ORM for the JavaScript programming language, and evaluate it on 8 JavaScript projects that use Sequelize. In these projects, REFORMULATOR found 44 instances of the "N+1 problem". Due to the highly dynamic nature of the JavaScript programming language, sound static analysis for JavaScript remains elusive [175, 130, 125], and as a result, it is possible for our implementation to propose refactorings that do not preserve behavior. Therefore, following other recent work on refactoring for JavaScript [100, 49], REFORMULATOR presents refactorings as *suggestions* that should be carefully vetted by a programmer, e.g., by running tests.

In practice, REFORMULATOR successfully refactored all 44 instances of the "N+1 problem", and in all cases performance was improved (up to 7.67x, even with small amounts of data being processed). Additional experiments revealed speedups of up to 38.58x and substantial improvements in scalability by demonstrating that the relative performance improvements grew as the database size was increased. We also confirmed that these performance gains translate to an improved user experience, by demonstrating reductions in page load times by up to 90% with large database sizes.

In summary, this chapter describes:

- an approach in which instances of the "N+1 problem" are detected by tracking data flow between ORM API calls, and where a set of declarative rewrite rules specifies how code can be refactored to eliminate them;
- an implementation of this approach in a tool called REFORMULATOR, targeting the popular Sequelize ORM in JavaScript;
- an evaluation of REFORMULATOR on 8 projects containing 44 instances of the "N+1 problem", demonstrating that the suggested refactorings improve performance and scalability, while preserving program behavior in all cases,

An artifact complete with the source code and the ability to re-run the experiments discussed in this chapter is available [215].

The remainder of this chapter is organized as follows. § 5.2 establishes relevant background via motivating example; § 5.3 details the approach to finding and refactoring "N+1 problem" anti-patterns; § 5.4 describes the implementation of this approach in a tool called REFORMULATOR; § 5.5 presents an evaluation of REFORMULATOR; § 5.6 identifies some threats to the validity of our approach; § 5.7 positions this work in the context of related literature; § 5.8 concludes; and finally § 5.9 presents a short retrospective of this

work, some promising follow-up work, and puts it into context with respect to the other work in this thesis.

5.2 Background and Motivation

To illustrate how "N+1 problem" issues arise in practice, consider **youtubeclone** [144], a popular open source video-sharing application emulating YouTube with over 125 stars and nearly 600 forks.

Like many database-backed web applications, the three components of **youtubeclone** are a front-end client-side interface, a back-end server, and a database. As users navigate through the front-end, HTTP requests are made to the server which sends HTTP responses once the requests have been processed. In some cases, the server will query the database if data is needed to prepare the response.

youtubeclone is written in JavaScript, and uses Sequelize [31], a popular ORM that enables JavaScript applications to interact with relational databases. The database backing **youtubeclone** has tables for videos, users, subscriptions, and views, and Figure 5.1 shows the Sequelize code modeling the video and user tables (simplified for brevity). The model corresponding to the video table is defined on lines 167-178, with the primary key "vid" defined on lines 168-173, and the model corresponding to the user table is defined on lines 179-190, with the primary key "uid" defined on lines 180-185. The association between the two models is made using a *foreign key*, i.e., a table column that contains the primary key of another table. Line 192 specifies "uploader" as a foreign key into the video table. This foreign key allows *joins* to be executed on the video and user tables, which fetches the user information associated with a video. E.g., a list of videos with "Alexi Thesis" in the title and information related to the uploader is obtained by the following Sequelize API call:

```
Video.findAll({include: {model:User},
  where: {[Op.substring]: {title: "Alexi␣Thesis"}}})
```

which would be translated into the following SQL query:

```
SELECT * FROM VIDEO LEFT JOIN USER ON USER.uid = VIDEO.uploader
  WHERE VIDEO.title LIKE "%Alexi Thesis%"
```

`Video.findAll` performs a `SELECT` from `Video` (since no attributes were specified, this is translated to `SELECT *`), `include` indicates that the generated query should include the

```
167 const Video = sequelize.define("Video", {
168   vid: {
169     type: DataTypes.UUID,
170     allowNull: false,
171     primaryKey: true,
172     defaultValue: Sequelize.UUIDV4,
173   },
174   title: {
175     type: DataTypes.STRING,
176     allowNull: false,
177   },
178 });
179 const User = sequelize.define("User", {
180   uid: {
181     type: DataTypes.UUID,
182     allowNull: false,
183     primaryKey: true,
184     defaultValue: Sequelize.UUIDV4,
185   },
186   username: {
187     type: DataTypes.STRING,
188     allowNull: false,
189   },
190 });
191 // Establish association between Video and User
192 Video.belongsTo(User, {foreignKey: "uploader"});
```

Figure 5.1: Example database definition in Sequelize.

```
195  async function recommendChannels(req, res)
         {
196    const channels = await User.findAll({
197      limit: 10,
198      attributes: ["id", "username", "avatar
       ", "channelDescription"],
199      where: { id: { [Op.not]: req.user.id }
         }
200    });
201   channels.forEach(async (channel, index)
       => {
202      const isSubscribed = await
       Subscription.findOne({
203        where: {
204          subscriber: req.user.id,
205          subscribeTo: channel.id,
206        },
207      });
208      channel.setDataValue("isSubscribed",
       !!isSubscribed);
209      // send HTTP response after processing
         the last channel
210    });
211  }
```

```
212  async function recommendChannels(req, res) {
213    const channels = await User.findAll({
214      limit: 10,
215      attributes: ["id", "username", "avatar",
        "channelDescription"],
216      where: { id: { [Op.not]: req.user.id } }
217    });
218    const subscriptions = await Subscription.
       findAll({
219      where: {
220        subscriber: req.user.id,
221        subscribeTo: channels.map(chan => chan
       .id)
222      }
223    });
224    channels.forEach(async (channel, index) =>
        {
225      const isSubscribed = subscriptions.find(
       data => data.subscribeTo === channel.id)
       ;
226      channel.setDataValue("isSubscribed", !!
       isSubscribed);
227      // send HTTP response after processing
       the last channel
228    })
229  }
```

Figure 5.2: **(a)** Functionality for recommending channels in *Youtube Clone*, exhibiting the "select N+1 problem". **(b)** Refactored version of the code, which generates fewer SQL queries.

associated `User` table by performing a `LEFT JOIN`, and `where` specifies that the query should only return videos with "Alexi Thesis" in the title.

SQL and many other query languages (and, by extension, Sequelize) also allows queries to specify a *grouping* clause, and *aggregations* over groups. If a query includes `GROUP BY ColumnName`, the results will be grouped according to unique values of `ColumnName`. Aggregate functions (such as `COUNT`) can be included in grouped queries, and the function is performed over the group. For example, the query `SELECT title, COUNT(title) FROM VIDEOS GROUP BY title` will yield all unique video titles as well as how many videos had that title.

To illustrate how ORMs may be misused, consider Figure 5.2(a), which shows some key fragments of a function `recommendChannels` from the back-end of **youtubeclone**. The function takes a parameter `req` representing a user request, and eventually produces an HTTP response that includes other channels that the current user (identified by `req.id`) might be interested in. This function first executes a call to `User.findAll` on lines 196–200 to determine a set of up to 10 channels for which the `id` is not the same as the current user

(i.e., the current user does not own the channel). This call is mapped by the ORM to a SQL query of the form `SELECT ··· FROM User LIMIT 10`.

Later, execution enters a loop (lines 201–210) that calls `Subscription.findOne(···)` to determine if the current user is already subscribed to each channel. Each of these calls is mapped by the ORM to an SQL query that looks as follows: `SELECT ··· FROM Subscription WHERE (Subscription.subscriber = ··· AND Subscription.subcribeTo = ···) LIMIT 1`. In other words, an initial query creates N results (here, $N = 10$) and *subsequently, a query is issued for each of these N results*, requiring a total of $N + 1$ database round-trips. The ORM community has recognized that, in such situations (referred to as the "N+1 problem"), it is often possible to modify the code to issue a lower, constant number of queries.

Figure 5.2(b) shows how the code of Figure 5.2(a) can be refactored to accomplish this. Here, an additional query is added on lines 218–223 to obtain an array `subscriptions` containing the channels from `channels` that the current user is subscribed to; on line 221, the `channels.map(...)` retrieves all of the `id`s for each `channel` so that the ORM can fetch the subscription status for all of the channels at once. In addition, in the loop over all channels (lines 224–228), the subscription status for a given channel is now looked up by calling the standard `find` method on arrays instead of querying the database. As a result, only 2 SQL queries are needed instead of the original N+1 queries.

`recommendChannels` contains two additional instances of the "N+1 problem" and both could be refactored similarly. The refactored code outperforms the original by a factor of nearly 3x.

Note the `await` on line 196: calls to Sequelize are asynchronous operations implemented using *promises* [11]. Refer to Chapter 2.2.3 for more information on promises.

The next section presents how the refactoring opportunities discussed in this section can be detected, and how code transformations can be generated automatically.

5.3 Approach

Our technique for suggesting refactorings that have the effect of eliminating the "N+1 problem" has two components:

1. a data flow analysis to locate pairs of ORM API calls involved in an "N+1 problem", discussed in § 5.3.1, and
2. a set of declarative rewrite rules describing how pairs of N+1-related ORM API calls are transformed to eliminate the problematic pattern, discussed in § 5.3.2.

5.3.1 Data-Flow Analysis

The main question the data-flow analysis is looking to answer is: *does data-flow exist between two ORM API calls?* Put differently, for every ORM API call C, the analysis should determine the existence of data-flow between the result of a previous ORM API call and any of C's arguments. This is achieved with a *taint analysis* [214, 121, 105], where ORM API calls are defined as *sources* of taint, and ORM API call arguments are defined as *sinks*. Concretely, we rely on a standard taint analysis framework available in CodeQL [152] to detect taint flows from sources to sinks.

For example, consider the code snippet in Fig 5.2(a). Here, the call to `findAll` returns a promise that will be resolved with the data from the database, and that value will flow into `channels`. Thus, there exists data-flow between `findAll` and `channels` through the promise created by `findAll`. The `forEach`-loop on lines 201-210 iterates over these values, and thus there is data-flow from elements of `channels` into the `channel` callback parameter (line 201). Finally, there is data-flow from `channel` into the argument of `Subscription.findOne` through the field access `channel.id` (line 205).

In order to generate code transformations, the approach needs the *property names* that are the target of data-flow (e.g., the analysis will report that data-flow exists between `subscribeTo : channel.id` and `channels`). Thus, the analysis notes exactly which property/value pairs $p : v$ in an ORM API call object O had values v that that were the target of data-flow from the result m of a previous ORM API call; in the following section, this process is encapsulated in the function $getAllPropertiesWithDataFlow(O, m)$.

5.3.2 Refactoring

Code transformations are presented as a set of declarative rewrite rules that can be found in Figure 5.3. The anatomy of the rules is:

$$\frac{conditions}{(\texttt{code before}) \rightsquigarrow (\texttt{code after})}$$

FINDALL-FINDONE This rule depicts the transformation for a flow from `findAll` through a loop into `findOne`. An example applying this rule to the code in Figure 5.2 follows this description.

1. First, the list of properties (*props*) of the argument to the `findOne` call ($\texttt{O}_2$) that are the targets of data-flow from the result of a call to `findAll` (`m1s`) is obtained through the helper function *getAllPropertiesWithDataFlow*.

$$\frac{\begin{array}{c}props = getAllPropertiesWithDataFlow(\texttt{O}_2, \texttt{m1s}) \\ \texttt{O}'_2 = updatePropReferences(props, \texttt{O}_2, \texttt{m1s}, \texttt{M}_1) \\ \texttt{BE} = createArrayLookup(props) \qquad \texttt{m2s}\ fresh\end{array}}{\begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{await M}_2\texttt{.}\underline{\texttt{findOne}}(\texttt{O}_2) \\ \}\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \texttt{var m2s} = \texttt{await M}_2\texttt{.}\underline{\texttt{findAll}}(\texttt{O}'_2) \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{m2s.find}(\texttt{m2} \Rightarrow \texttt{BE}) \\ \}\end{array}}$$

(FINDALL-FINDONE)

$$\frac{\begin{array}{c}props = getAllPropertiesWithDataFlow(\texttt{O}_2, \texttt{m1s}) \\ \texttt{O}'_2 = addAggregationAndCount(props, \texttt{O}_2, \texttt{m1s}, \texttt{M}_1) \\ \texttt{BE} = createArrayLookup(props) \qquad \texttt{m2s}\ fresh\end{array}}{\begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{await M}_2\texttt{.}\underline{\texttt{count}}(\texttt{O}_2) \\ \}\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \texttt{var m2s} = \texttt{await M}_2\texttt{.}\underline{\texttt{findAll}}(\texttt{O}'_2) \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{m2s.find}(\texttt{m2} \Rightarrow \texttt{BE})\texttt{.count} \\ \}\end{array}}$$

(FINDALL-COUNT)

$$\frac{\begin{array}{c}\exists\ dataFlow(m1s, x) \qquad \texttt{pk primary key of M2} \\ \texttt{O}'_2 = \{\texttt{where} : \{\texttt{pk} : \texttt{m1s.map}(\texttt{m1} \Rightarrow \texttt{m1.f})\}\} \qquad \texttt{m2s}\ fresh\end{array}}{\begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{await M}_2\texttt{.}\underline{\texttt{findByPk}}(\texttt{x.f}) \\ \}\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \texttt{var m2s} = \texttt{await M}_2\texttt{.}\underline{\texttt{findAll}}(\texttt{O}'_2) \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{m2s.find}(\texttt{m2} \Rightarrow \texttt{x.f} == \texttt{m2.pk}) \\ \}\end{array}}$$

(FINDALL-FINDBYPK)

$$\frac{\begin{array}{c}props = getAllPropertiesWithDataFlow(\texttt{O}_2, \texttt{m1s}) \\ \texttt{O}'_2 = updatePropReferences(props, \texttt{O}_2, \texttt{m1s}, \texttt{M}_1) \\ \texttt{BE} = createArrayLookup(props) \qquad \texttt{m2s}\ fresh\end{array}}{\begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{await M}_2\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_2) \\ \}\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\texttt{var m1s} = \texttt{await M}_1\texttt{.}\underline{\texttt{findAll}}(\texttt{O}_1) \\ \texttt{var m2s} = \texttt{await M}_2\texttt{.}\underline{\texttt{findAll}}(\texttt{O}'_2) \\ \texttt{loop } \{ \\ \quad \texttt{var m2} = \texttt{m2s.filter}(\texttt{m2} \Rightarrow \texttt{BE}) \\ \}\end{array}}$$

(FINDALL-FINDALL)

Figure 5.3: Declarative rewrite rule definitions. ORM API calls are underlined—these calls generate queries. The calls to `find` in the refactored code are essentially maps over the arrays `m2s` that return the element matching the boolean expression specified in the callback. Helper function descriptions can be found in § 5.3.3. All are discussed in detail in § 5.3.

2. The goal of this transformation is to insert a new ORM API call to `findAll` replacing the old call to `findOne`, and so the argument to that new call must be constructed. The idea is adapt the argument to the old call ($\texttt{O}_2$); since the new call will be placed *before* the loop, any properties in $\texttt{O}_2$ that were targets of data-flow must be updated to map directly over the result of the previous API call (`m1s`).

   To achieve this, a new object $\texttt{O}'_2$ is adapted from $\texttt{O}_2$ by updating all of the values of the properties in $\texttt{O}_2$ referred to by *props* to be maps over `m1s`, through the *updatePropReferences* helper function. For all properties $p : v$ in *props*, the property `f` of the model `M1` referred to by `v`, either directly in `v` itself (e.g., if `v` is of the form `x.f`) or indirectly (e.g., if $\texttt{v} = \texttt{x.f}$ earlier in the code) is obtained, and v is replaced with $\texttt{m1s.map(m1} \Rightarrow \texttt{m1.f)}$ in $\texttt{O}'_2$

3. As the goal of this refactoring is to replace many calls to `findOne` with a single call to `findAll`, the result `m2s` of that new call will need to be iterated over to pick out the same data that was returned by the original call to `findOne`. `m2s` contains all of the data that would have been fetched in the loop, and the idea here is to map whatever comparisons were being made in the original call to `findOne` to some new boolean expression (`BE`) that can be used to pick out the datum of interest from the array of results (`m2s`). This is achieved through the *createArrayLookup* helper function: for each property/value pair $p : v$ in *props*, a boolean expression $\texttt{m1.p} === \texttt{v}$ is added to `BE` (here, `m1` is the parameter name of a callback that will be inserted by the transformation). In constructing `BE` in this manner, the same comparisons that were being made in the old `findOne` are performed in `BE`.

4. To enact the transformation, a fresh variable `m2s` is declared and set to the return value of a new call to $\texttt{M2.findAll}(\texttt{O}'_2)$, and is placed immediately before the loop; the old call to $\texttt{M2.findOne}(\texttt{O}_2)$ is replaced with a lookup over the `m2s` array, and the entry matching `BE` is picked out.

FINDALL-FINDONE **(Walk-through)** To help illustrate the rewrite rule, consider the transformation in Figure 5.2.

1. First, there is data-flow between `channels` and the argument to `Subscription.findOne` in the `subscribeTo:` `channel.id` property; mapping to the FINDALL-FINDONE rewrite rule, this property will be the sole element of *props*.

2. The new ORM API call object (lines 218-223) is obtained from the existing call object (lines 202-207), where the value of the property with data flow (`subscribeTo:`

`channel.id`) is updated to map over `channels` (`channels.map(chan => chan.id)`; this is $\texttt{O}'_2$.

3. A new boolean expression `BE` is built from the properties that had data from `channels` flow into them, in this case the sole property with data flow `subscribeTo: channel.id` populates `BE` with the boolean expression `data.subscribeTo === channel.id`.

4. Putting it all together: the new call to `Subscription.findAll` is placed before the loop (lines 218-223), and the old call to `Subscription.findOne` is replaced with a `find` over the array of subscriptions returned by `Subscription.findAll` (line 225).

FINDALL-COUNT This rule depicts the transformation for data-flow into a call to `count`. The list of properties with data flow from `m1s` is obtained with *getAllPropertiesWithDataFlow* as in FINDALL-FINDONE. The new ORM API call object $\texttt{O}'_2$ is created in much the same way as well, except that in this case grouping and aggregation is added to $\texttt{O}'_2$: each property name referred to in *props* is added to a grouping clause in $\texttt{O}'_2$, and also to a count aggregation over those same properties (and that count is saved on the "count" field of the result). I.e., the results of the new call to `findAll` will be grouped by the properties with data flow, and total counts will be computed for each group. The rest of the rewrite rule is the same as FINDALL-FINDONE, except that the new access in the loop also specifies that the count field should be accessed.

For an example of this transformation, consider the snippets in Figure 5.4. There is data flow from the video `id` property to the view `videoId` property (line 243), and so the transformed code includes a grouping clause on `videoId` (line 264), and count over `videoId` as well (line 266). To break it down further, the Sequelize line `[Sequelize.fn("COUNT", Sequelize.col("View.videoId")), "count"]` is specifying that a count over `View.videoId` should be issued, and saved under the `count` property of the result. That property is referenced in the loop in the transformed code, on line 269.

FINDALL-FINDBYPK Calls to `findByPk` take a single argument that is implicitly compared against the primary key of the model being queried. That implicit comparison needs to be made explicit in the new `findAll` query, and so the primary key `pk` of model `M2` is obtained from the model definition. Then, the new call object O'_2 can be constructed with a `where` clause that compares the primary key `pk` with a map over the sources `m1s` extracting the relevant field `f` (i.e., the field from the data-flow into the call to `findByPk`). The primary key `pk` is also needed to construct the boolean expression in the find that replaces the old call to `findOne`.

```
230 exports.searchVideo = asyncHandler(async (req, res, next) => {
231     const videos = await Video.findAll({
232         include: {
233             model: User,
234             attributes: ["id", "avatar", "username"]
235         },
236         where: {
237             title: {
238                 [Op.substring]: req.query.searchterm
239     }}});
240     videos.forEach(async (video, index) => {
241         const views = await View.count({
242             where: {
243                 videoId: video.id
244             }
245         });
246         // ...
247     });
248 });
249
```

```
250 exports.searchVideo = asyncHandler(async (req, res, next) => {
251     const videos = await Video.findAll({
252         include: {
253             model: User,
254             attributes: ["id", "avatar", "username"]
255         },
256         where: {
257             title: {
258                 [Op.substring]: req.query.searchterm
259     }}});
260     const viewCounts = await View.findAll({
261         where: {
262             videoId: videos.map(data => data.id)
263         },
264         group: ["View.videoId"],
265         attributes: ["videoId", [Sequelize.fn("COUNT",
266             Sequelize.col("View.videoId")), "count"]]
267     });
268     videos.forEach(async (video, index) => {
269         const views = viewCounts.find(x => x.videoId === video.id).count;
270         // ...
271     });
272 });
273
```

Figure 5.4: **(a)** Functionality for search for a video in **youtube-clone**, where the views for each video are counted in the loop. **(b)** Refactored version of the code, which generates fewer SQL queries. Note the grouping clause on line 264, and the count attribute on line 266 which sums up the number of elements in each group.

FINDALL-FINDALL Finally, this rule is nearly identical to the FINDALL-FINDONE rule, the only difference is that instead of performing a `find` over the `m2s` array, a `filter` is performed instead.

Note The idea that data-flow between ORM API calls is problematic is language-agnostic, and while the rewrite rules use Sequelize API names in them, that is more for readability; the rules represent broader issues in ORMs like finding and then finding again (FINDALL-FINDONE, FINDALL-FINDALL, FINDALL-FINDBYPK), or finding and then counting (FINDALL-COUNT). This is essential functionality to any effective ORM.

5.3.3 Helper Function Reference

This section contains in-depth descriptions of the helper functions used in Fig. 5.3.

getAllPropertiesWithDataFlow(O, m) returns all of the properties in an object O that are targets of data-flow from some value m. This will yield a set of property name, value pairs `p : v` for which there exists data-flow between m and the value `v`.

updatePropReferences$(props, O, ms, M)$ creates an object where all of the properties in an object O specified by the list of properties *props* are updated to refer to a map over the array ms. I.e., for all property/value pairs `p : v` in *props*, the matching property in O' will be `p : ms.map(m => m.f)`, where `f` is the property of the model M referred to by `v`, either directly in `v` itself (e.g., if `v` is of the form `x.f`) or indirectly in some alias (e.g., if `v` = `x.f` earlier in the code).

addAggregationAndCount$(props, O, ms, M)$ creates a new object wherein all of the properties in the object O specified by the list of properties *props* are updated to refer to a map over the array ms, like *updatePropReferences*. Additionally: (1) a clause is added grouping by all property names p in *props*, and (2) a count aggregation clause is added to total the number of entries in each group.

createArrayLookup$(props)$ builds a boolean expression `BE` to select from the array of results the value that was previously obtained by the query. A property `p : v` has the ORM compare the value of `v` against property `p`, and so a boolean expression `m1.p == v` is created and added with a boolean & to BE.

5.4 Implementation

The approach described in § 5.3 is implemented in a tool called REFORMULATOR. The static data flow analysis is implemented as a taint analysis in CodeQL [152], wherein a taint configuration [153] specifies values returned by ORM API calls as sources, and arguments passed to ORM API calls as sinks. The rewrite rules were implemented using BabelJS [52], a popular JavaScript parser and code generator. Taint flows identified by the analysis are input to the refactoring tool. Sound and scalable static analysis of JavaScript is beyond the current state-of-the-art, and so the code transformations generated by REFORMULATOR are presented to the programmer as suggestions that should be vetted carefully, e.g., by running tests. The code is available in the accompanying artifact [215], which is a Docker image equipped with the ability to re-run the entire evaluation, which is discussed next.

5.5 Evaluation

This evaluation of REFORMULATOR aims to answer the following research questions:

RQ1. How many refactoring opportunities does REFORMULATOR detect?

RQ2. How often are unwanted behavioral changes introduced by the refactorings suggested by REFORMULATOR?

RQ3. How do the refactorings affect performance?

RQ4. How much do the refactorings affect page load times?

RQ5. What is the running time of REFORMULATOR?

Experimental Setup We randomly selected 100k JavaScript GitHub repositories that listed Sequelize as an explicit dependency. We then ran the `npm-filter` [50] tool on these repositories to determine how many of them could be automatically installed and built; 37,074 projects satisfied these criteria. We then ran the CodeQL taint analysis on these projects and found 427 projects with N+1 anti-pattern query pairs. From those, we randomly selected projects until we found 8 that we could set up and run with databases populated with meaningful data. Project statistics are listed in Table 5.1.

Table 5.1: Information about subject applications. The first row reads: *the first application is called* ***youtubeclone****, and commit hash* `47002fc` *was used for the evaluation;* ***youtubeclone*** *has 10,551 lines of code spread across 117 files.* REFORMULATOR *detected 12 N+1 pattern query pairs in this application across 7 HTTP request handlers. This is a video-sharing application.*

Project Name	Commit Hash	LOC	Num. Files	Num. N+1	Num. Handlers	Short Description
youtubeclone [144]	`47002fc`	10,551	117	12	7	Video sharing.
eventbright [219]	`e417020`	12,085	122	15	7	Event search and attendance.
property-manage [155]	`33f92a9`	13,959	154	2	2	Property management app.
Math_Fluency_App [180]	`5c1658e`	12,473	114	6	3	Math testing for teachers.
employee-tracker [65]	`ba4a195`	10,336	112	3	2	Human resources server API.
Graceshopper-Elektra [76]	`c327530`	12,342	141	1	1	Shopping application.
wall [34]	`ae6c815`	11,152	134	2	2	Image hosting and tagging app.
NetSteam [207]	`5b1cd86`	12,485	136	4	4	Video game trailer viewing app.
			Sum	**44**	**27**	

Experiment Infrastructure Experiments were conducted on a 2016 MacBook Pro with 16GB RAM and 2.6 GHz Quad-Core Intel Core i7 processor running MacOS Catalina v10.15.7. The Chrome browser v100.0.4896.127 was used in incognito mode so as to minimize interference from caching and browser extensions.

RQ1: How many refactoring opportunities does REFORMULATOR detect?

To answer this research question, we examined the number of projects in which REFORMULATOR identified anti-patterns. Overall, 427 contained at least one instance of an N+1 anti-pattern from those that built. We examined the distribution of N+1 anti-patterns across the projects; the median number of anti-patterns is 2, and a total of 1,872 anti-pattern instances were detected by the tool. While this is not a huge percentage of the projects (1.1%), the analysis is quite conservative in order to maximize the likelihood of the transformation succeeding.

REFORMULATOR identified refactoring opportunities in hundreds of GitHub repositories.

RQ2: How often are unwanted behavioral changes introduced by the refactorings suggested by REFORMULATOR?

To answer this research question, we identified which HTTP request handlers in each of the projects contained a refactoring opportunity detected by REFORMULATOR. Every refac-

toring suggestion was applied to the code. We focused on these handlers as they are the manner in which a front-end would interact with the server; if the handler produces the same response, we deem the behavior to be preserved. There were 44 refactoring opportunities spread across 27 handlers as outlined by columns **# N+1** and **# Handlers** in Table 5.1. The FINDALL-FINDALL rewrite rule was applied 10 times, FINDALL-FINDBYPK 9 times, FINDALL-FINDONE 5 times, and FINDALL-COUNT 20 times. Note that for this experiment, the databases were populated with test data according to the instructions provided by the repositories.

To conduct the experiment, the UI for each page issuing the HTTP requests and the actual content of the HTTP response was closely examined and compared before and after refactoring. No discrepancies were found, and no refactoring introduced a crash.

REFORMULATOR did not introduce any unwanted behavioral changes in the applications we studied.

RQ3: How do the refactorings affect performance?

To answer this research question, we inserted profiling code in the aforementioned HTTP request handlers to collect the time it took the server to prepare a response. We manually interacted with the front-end of each of the subject applications to locate the part of the front-end that sent the request triggering the anti-pattern code. We then restarted the server to empty any server-side caches, triggered the HTTP request again, and collected the time reported by the aforementioned profiling code. We repeated this process ten times before applying the code transformations, and ten more times after: averages and standard deviations of these results are reported in Figure 5.5 (the error bars represent the average +/- one standard deviation), with each pair of bars corresponding to the time before and after refactoring for a particular HTTP request handler. There are 27 total pairs of bars, corresponding to each of the affected HTTP request handlers, and a link from each "HTTP Request Handler ID" to the code is included in Appendix B.

We found that a low, constant number of queries were issued post-refactoring in all cases, and that every refactoring improved performance. Specifically, we performed a paired two-tailed T-test comparing the 10 run times before and after at 95% confidence and found all differences to be statistically significant. The largest performance gain was in **eventbright**'s handler for getting all events (ID 10, from 279.77ms before to 36.48ms after, an improvement of 7.67x). All HTTP request handlers in **youtubeclone** (IDs 0 through 6) had pronounced improvements, with a median performance improvement of a

Table 5.2: Information about the run time of REFORMULATOR, with project installation time given for reference. The first row of the table reads: ***youtubeclone*** *took 5.42s to install; it took 24.96s to build the CodeQL database; it took 30.10s to run the N+1 detection query. In total, from a freshly installed* ***youtubeclone****,* REFORMULATOR *can run in 55.06s.*

Project Name	Install Time (s)	QLDB Build Time (s)	Query Run Time (s)	Build + Query (s)
youtubeclone [144]	5.42	24.96	30.10	55.06
eventbright [219]	11.42	28.64	32.39	61.03
property-manage [155]	14.68	30.91	33.17	64.08
Math_Fluency_App [180]	4.87	24.41	33.62	58.03
employee-tracker [65]	4.20	23.43	29.41	52.84
Graceshopper-Elektra [76]	24.29	26.69	30.33	57.02
wall [34]	17.29	26.35	29.88	56.23
NetSteam [207]	14.50	29.02	31.79	60.83
Mean	**12.08**	**26.80**	**31.34**	**58.14**

factor of 2.81x. The smallest benefits were in the **Math_Fluency_App** application (IDs 17 through 19), with a median improvement factor of 1.07x—this is because the number of queries was very small even before refactoring (the number of queries was reduced from 5 to 3, as N was small for this application).

To further understand the performance implications of the refactorings, particularly as database size increased, we conducted a case study involving five request handlers from the 27 in which we refactored instances of the "N+1 problem". In this case study, we created three databases of size 10, 100, and 1000 (henceforth referred to as the "10 scale", "100 scale", and "1000 scale" configurations) so that the HTTP request handler needs to process that much data, and measure the performance of the handlers before and after refactoring at each database size.

The functionality being examined in each application is:

- **youtubeclone**: search for users;
- **eventbright**: main events display;
- **property-manage**: properties dashboard;
- **employee-tracker**: view all employees;
- **NetSteam**: view all reviews for a trailer.

The results of this case study are summarized in Table 5.3, which reports averages over 10 runs for each database size for each request handler. **youtubeclone**, **eventbright**, and

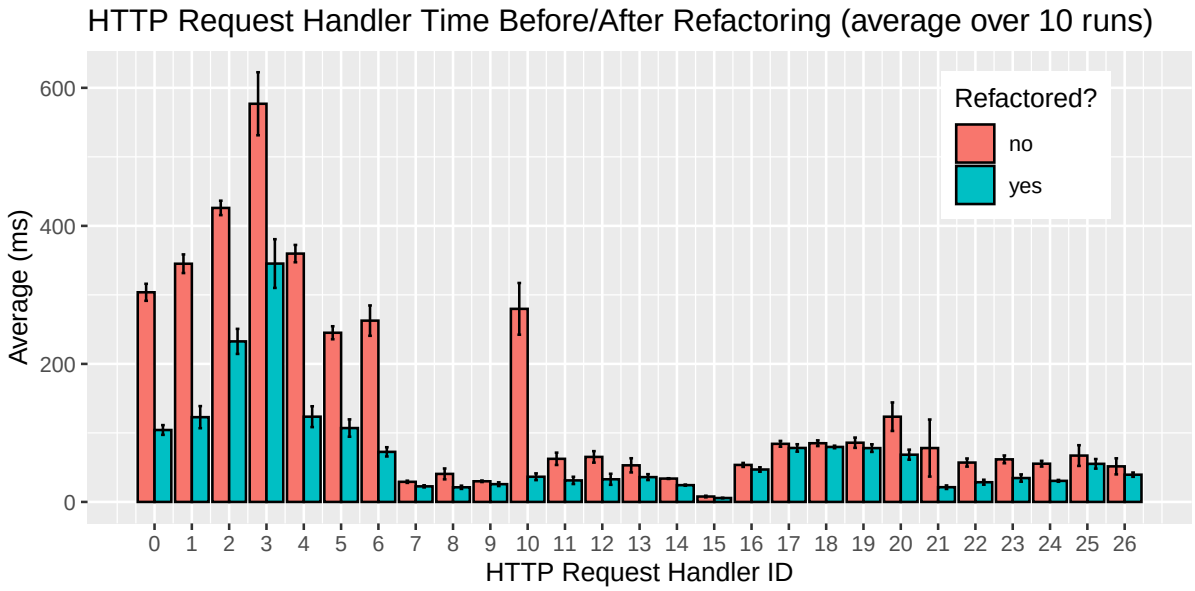


Figure 5.5: Summary of effect of refactoring on 27 HTTP request handlers. Lower is better. Each pair of bars corresponds to an HTTP request handler. Error bars indicate +/- one standard deviation.

Table 5.3: Results of case study on 5 applications comparing the scalability of original and refactored code. All times are in ms. The differences were all statistically significant (paired two-tailed T-test at 95% confidence); standard deviations are omitted for brevity, and can be found in supplemental material. The first row of the table reads: *for test ID 1 in the* ***youtubeclone*** *application, with a database size of 10, the mean before refactoring is 360.30ms, and after refactoring is 118.06ms; this represents a performance improvement with a factor of 3.05x (= 360.30 ÷118.06).*

		DB Size = 10			***DB Size = 100***			***DB Size = 1000***		
Project Name	**ID**	**Before**	**After**	***Scale***	**Before**	**After**	***Scale***	**Before**	**After**	***Scale***
youtubeclone	1	360.30	118.06	*3.05x*	1937.42	152.96	*12.67x*	18171.86	471.07	*38.58x*
eventbright	10	111.38	31.94	*3.49x*	797.35	49.53	*16.10x*	7001.48	214.61	*32.62x*
property-manage	20	56.91	33.71	*1.69x*	246.06	111.05	*2.22x*	1333.64	786.44	*1.70x*
employee-tracker	14	57.15	34.32	*1.67x*	374.73	153.97	*2.43x*	2495.92	1010.47	*2.47x*
NetSteam	21	77.05	39.01	*1.98x*	337.67	41.62	*8.11x*	2129.34	108.06	*19.71x*

NetSteam show dramatic improvements in the relative performance benefits of the refactored code as databases size increases (up to *38.58x* at the 1000 scale for **youtubeclone**). In contrast, the relative performance difference for **property-manage** and **employee-tracker** is not as pronounced with large database sizes; in these applications, most of the time spent serving requests is in processing the data from the database once it is available, rather than waiting for it to become available. Nevertheless, the absolute difference between original and refactored code is substantial at large database sizes even for those two applications, with a 550ms difference for **property-manage** and a nearly 1.5s difference for **employee-tracker**.

All transformations yield statistically significant performance improvements at 95% confidence. Performance gains increase as the size of the database grows; we observed speedups of up to 38.58x.

RQ4: How much do the refactorings affect page load times?

In this research question, we aim to connect the performance improvements observed in serving HTTP requests to measurable improvements in page load time on the client-side.

We conducted a case study on the client-side pages making the HTTP requests studied in the context of **RQ3**. Note: there is no front-end for **employee-tracker**, thus we focus on the other four. The manner in which pages load varies significantly from one application to another, and we found no reliable way to universally time each page load. For example, the **NetSteam** page under study is a pop-up that displays over the main dashboard, and has no URL associated with it, making refresh-based profiling impossible. Further, most web profiling tools rely on the collection of a trace as a page loads, and that trace includes a variable number of frames *before* the page begins to refresh, leading to unfortunate variability and inaccuracies in performance numbers collected automatically.

In light of this, we opted to manually study the behavior of each page with Chrome DevTools [102] to obtain rich information about how each page behaves, paying particular attention to the "Performance" and "Network" tabs. The times reported are estimations based on the trace timeline displayed by the "Performance" tab of the Chrome DevTools (label (E) in Figure 5.6) that displays a timeline of screenshots of a page, which we believe corresponds most closely with the observable user experience. Specifically, as in our study of **RQ3**, we triggered each HTTP request 10 times and estimated the time between when the request was triggered and when the page was visibly populated with data; we drew these estimates from the time markers in the timeline, and rounded to the nearest quarter

second, and averages are reported throughout this section. We examined the behavior of each page at three database scales (10, 100, and 1000), and report on our findings below. Screenshots of the DevTools profiles used in this study as well as raw observations are included in Appendix B.

youtubeclone (search for users) In this application, we found the network time to be the limiting factor in the client page being fully rendered, as the page was quickly populated once all the data was returned from the server. At the 1000 scale, the difference in load time was dramatic (19.88s with the original code vs. 1.9s with the refactored code, a ∼10x improvement). The difference in load time is also very noticeable at the 100 scale, and a screenshot comparing the effect of the transformation on the load time can be found in Figure 5.6 (3.8s before refactoring vs. 0.8s with refactoring). Even at the smaller 10 scale, appreciable load time improvements were observed (from 1.2s to 0.5s).

eventbright (main events display) The front-end is quickly populated with data once it is received from the server. We noted dramatic load time improvements at the 1000 scale (7.7s with original code vs. 1.4s with refactored code), and a noticeable improvement at the 100 scale (1s with original code vs. 0.3s with refactored code), and a very small difference at the 10 scale (0.4s before vs. 0.3s after).

property-manage (property dashboard) In this application, the refactoring did not appear to affect the load time of the page. Even at the 1000 scale, the dashboard took nearly 3s to be populated with data, even though the server finished fully processing the request 1.5s faster in the refactored version. This is because the information computed by the ORM API call in the loop is used internally by the server, and is not part of the response.

In spite of this, the refactoring is still beneficial: as applications move away from locally-hosted databases, the number of concurrent database requests becomes a concern, as many remote database management systems only allow up to a certain number of requests simultaneously, after which point requests are refused. The refactoring proposed by REFORMULATOR reduces the number of requests here from N+1 (with N being the number of properties) to two.

NetSteam (reviews for trailer) Here, a dashboard presents many video game titles to the user, and the user may select one of them to bring up an animated pop-up with

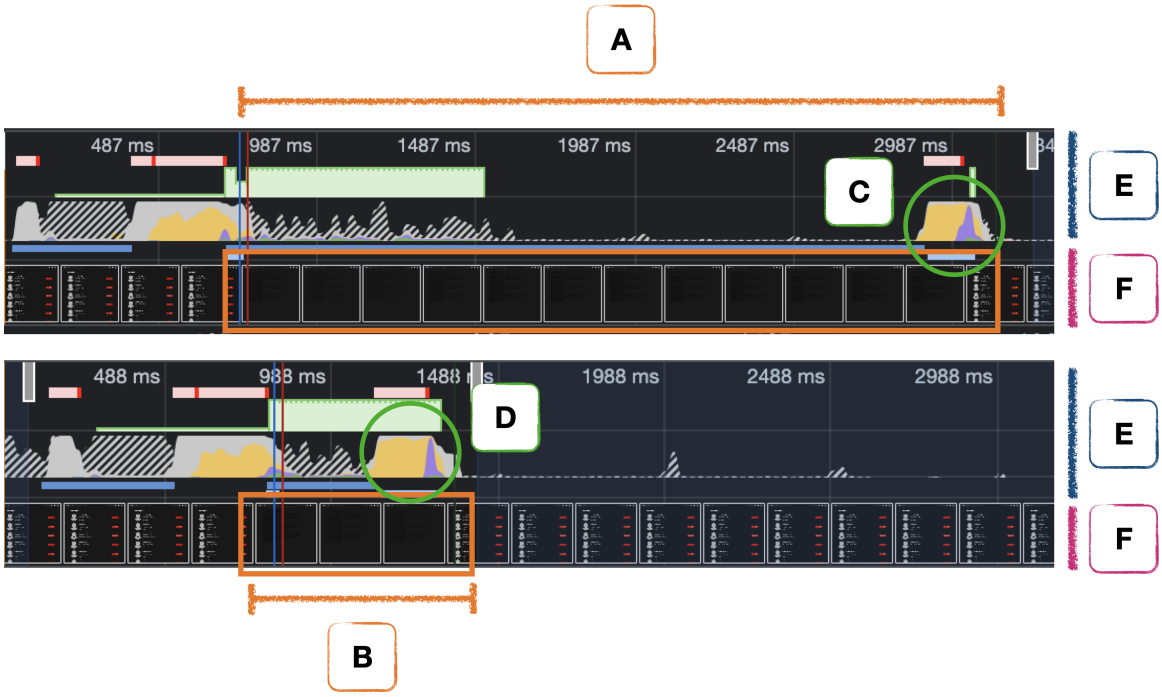


Figure 5.6: Two screenshots from the Chrome DevTools' Performance Tab profiling a search turning up 100 users in **youtubeclone**. The profile corresponding to the original code is on top, and the refactored one is on the bottom. The two (E) labels show time series of application activity, where higher values correspond to more CPU cycles. (C) and (D) show spikes in activity when the HTTP response was received by the client before and after refactoring, resp. The two (F) labels show a series of screenshots taken of the front-end as it loads and is populated by data. (A) and (B) show the period that the screen was idle before and after refactoring, resp., and the two boxes in the timelines highlight that the screen is empty during that span.

the trailer and reviews for the game. At the 1000 scale, it took 3.8s on average for the reviews to load with the original code vs. 2s with the refactored code. At the 100 and 10 scales, the animation displaying the trailer and reviews masked any performance difference between original and refactored code, as the animation completes before the reviews load at both scales before and after refactoring.

In several cases, the refactoring suggested by REFORMULATOR results in dramatic speedups (of up to 90%).

RQ5: What is the running time of REFORMULATOR?

Table 5.2 shows the time it takes `npm install` to install the project's dependencies (given for reference, column **Install Time**), the time it takes to build the CodeQL database, which is needed to run any CodeQL queries on the code (**QLDB Build Time**), and the time to run REFORMULATOR's anti-pattern detection query (**Query Run Time**). The time taken to build the QLDBs *and also* run the queries is consistently between 50 and 65 seconds. The time to run the actual code transformation is less less than a second in all cases and is not reported in the table.

The running time of REFORMULATOR on a fresh installation of a project is 58.14s on average.

5.6 Threats to Validity

We have identified some threats to the validity of our work.

The primary threat to validity is the fact that the transformations proposed by our tool may not preserve program behavior. Static analysis of JavaScript is unsound due to the extreme dynamicity of the language, as rampant dynamic property redefinition, event-driven programming, and promise-based asynchrony have made precise and scalable analysis elusive. REFORMULATOR is a tool that leverages static program analysis, and is thus unsound; we have accepted this in designing REFORMULATOR, and focused on developing a tool that is practical. During the course of our evaluation, we found that no behavior-altering transformations were suggested.

It is also possible that our selection of projects for evaluation is not representative. We mitigate this by selecting projects randomly from those that explicitly declare Sequelize as

a dependency. This list was pruned to find projects that could be successfully built and for which we could configure and populate databases, but this was entirely so that the effect of the transformations could be studied.

5.7 Relation to Previous Work

There is a large body of existing research aimed at improving the performance of database-backed applications, including database refactoring, bug detection, and query optimization.

Database refactoring. Existing work has considered refactoring database schemas to improve performance. Ambler and Sadalage [44] catalogue *database refactorings*, i.e., behavior-preserving changes to a database schema such as moving a column from one table to another. Similarly, Xie et al. [238] and Wang et al. [230] study how application code must be updated in response to schema changes. Rahmani et al. [179] present an approach for avoiding serializability violations in database applications by transforming a program's data layout. This nature of work provides insight into the relationship between database structure and performance, but does not consider query-based performance bugs like the "N+1 Problem".

Identifying the "N+1 Problem" in database code. Yang et al. [240] use dynamic analysis to detect performance anti-patterns in Ruby on Rails [187] applications and manually refactor them to assess performance impact. One of these anti-patterns, "inefficient lazy loading", is a variant of the "N+1 Problem" they report to be prevalent in their experiments. Chen et al. [58] report on industrial experience, observing 17 ORM-related performance problems in PHP applications that use the Laravel ORM [30], including the same "inefficient lazy loading" anti-pattern. Chen et al. [59] use static analysis to detect anti-patterns in JPA, a popular ORM for Java, including "one-by-one processing" where a list of objects of one class is iterated over, and objects from another class are found by issuing a SELECT query. Their proposed resolution involves introducing *batching* (i.e., waiting for several queries to be created before issuing them all at once). Cheung et al. [61] created a "lazy-ifying" compiler that also batches queries to reduce the number of round trips to the database. Batching queries does alleviate the "N+1 Problem" by reducing the amount of database round-trips, but it does not eliminate the problem through permanent refactoring. Also, much prior work [240, 58, 59] detects the "N+1 Problem" but does not automatically refactor it as we have in REFORMULATOR.

Identifying other performance bugs in database code. Chen et al. [60] consider situations where calling the API of the Hibernate ORM [108] for Java results in accessing

redundant data (e.g., some columns in a table need to be updated, but a query is generated that updates *all* of them). They assess performance impact by manually rewriting subject applications. Yan et al. [239] identify optimization opportunities in Ruby on Rails [187] applications using static analysis and profiling, including a "Fusing queries" optimization targeting situations where the result of a query flows into another query. Yang et al. [242] present a framework in which static analysis and dynamic profiling are used to visualize, for each HTML tag, the set of database queries needed to generate the data needed to render it. Their framework also suggests view-changing refactorings (e.g., introducing pagination) to improve performance. While there is much work on detecting query-based performance bugs, including the "N+1 Problem", using static and dynamic analysis, this work leaves actual optimization to manual refactoring.

We know of two research efforts to use static analysis to automatically refactor source code to remove database bugs. Yang et al. [241] design a RubyMine IDE plugin named PowerStation which uses static analysis to identify and refactor common ORM performance inefficiencies. While this work relates most closely to ours, PowerStation does not identify or refactor the "N+1 Problem". Instead, PowerStation tackles other inefficiencies like dead stores, redundant loads, and Ruby-specific API misuses. Lyu et al. [139] present an automatic refactoring technique for repetitive autocommit transactions, using static analysis to detect this database ineffiency common to the Android platform. However, repetitive autocommit transactions refer to writes, whereas the "N+1 Problem" concerns reads.

In sum, previous work explored database-related refactorings and the detection of ORM anti-patterns. However, we are not aware of automated refactoring tools for eliminating the "N+1 Problem".

5.8 Conclusion

ORMs provide an object-oriented interface to databases and facilitate the development of database-backed applications. In an ORM, databases can be accessed using method calls to the ORM, which maps those calls into database queries. While convenient, this added layer of abstraction hides the significant performance cost of database operations, and misuse of ORMs can lead to far more queries being generated than necessary. In particular, the "N+1 problem" is prevalent in ORM-backed applications. It is natural to iterate over collections in object-oriented languages, but iterating over data that originates from a database and calling an ORM method in each iteration may result in suboptimal

performance. In such cases, it is often possible to reduce the number of round-trips to the database by issuing a single query that fetches all desired results at once.

In this work, we presented an approach for automatically refactoring applications that use ORMs to eliminate instances of the "N+1 problem", which relies on static analysis to detect data flow between ORM API calls. We implemented this approach in REFORMULATOR, a tool targeting the Sequelize ORM in JavaScript, and evaluated it on 8 JavaScript projects. We found 44 N+1 query pairs in these projects, and REFORMULATOR refactored all of them successfully, resulting in improved performance while preserving program behavior. At a small scale, performance improvements of up to 7.67x were observed, and improvements of up to 38.58x were observed at scale. Further, a detailed study of the front-ends of these applications revealed page load time improvements of up to 90%.

5.9 Discussion

In this chapter, we proposed an approach to automatically detecting and repairing instances of the "N+1 Problem" in ORM-backed applications. We implemented this approach in REFORMULATOR, which relies on an unsound static data flow analysis to identify pairs of data-related ORM API calls as candidates for refactoring. This goes one step further than *DrAsync* in suggesting fixes automatically, enabled in part by the narrower scope of this work; *DrAsync* was concerned with general anti-patterns, while this work is focused on a specific misuse of ORM APIs in which far more precise information available. First, source API calls always return arrays of objects with a predictable shape: a call like `Video.findAll(...)` will return a `Video` array, and the shape of `Video` is known thanks to the statically available ORM model files. Further, ORM API calls are very strict in the shape of their arguments: in Sequelize and TypeORM, for instance, calls to the ORM are supplied with objects who's properties are either from a pre-defined set or correspond to columns in the underlying database. E.g., in `Video.findAll({where: { name: "Alexi Thesis Defense", length: 1800 }})`, `name` and `length` are properties of the `Video` model, and `where` specifies a *where* clause in the generated query. These are just a few sources of information that imprecise analysis can take advantage of. (The narrower scope also benefited the approaches that will be discussed in Chapters 6 and 7, where automated program transformations were feasible.)

For a bit of history, we initially identified pairs of data-related ORM API calls via dynamic analysis, but found that the vast majority of the data-related ORM API calls *that could be refactored* were intraprocedural and good candidates for detection through lightweight static analysis instead. Besides that, additional information was required in

order to automate refactoring (e.g., variable names, parsing the ORM API calls to determine information about the query, etc.), and so static analysis was required anyway. The main advantage of a dynamic vs. static approach would be a substantial increase in precision, although that precision would be wasted here since (a) there is plenty of information available for imprecise analysis, and moreover (b) interprocedural data-related ORM API calls are quite complex to refactor.

The work in this chapter focused on the "N+1 Problem", but *most* data-related ORM API calls are unnecessary as databases are well-equipped to resolve relationships. We observed many situations where non-N+1 related ORM API calls could be optimized:

```
const user = await User.findByPk(req.params.id);

if (!user) {
  return next({
    message: `No user found for ID - '${req.params.id}'`,
    statusCode: 404,
  });
}

const isSubscribed = await Subscription.findOne({
  where: {
    subscriber: req.user.id,
    subscribeTo: req.params.id,
  },
});
```

In this snippet, the programmer determines if the requesting user (identified by `req.user.id`) is subscribed to another user (identified by `req.param.id`). Here, `req.params.id` is used to select both a `User` and a `Subscription`. From the models, we know that `User` has a *has many* relationship with `Subscription` through the `subscribeTo` foreign key, so all `Subscription.subscribeTo` values will be `User` primary keys. Given that, we can reduce the number of ORM API calls and database round trips by:

```
const user = await User.findByPk(req.params.id, {
  include: [
    {
      model: Subscription,
      required: false,
      where: {
        subscriber: req.user.id,
        subscribeTo: req.params.id
      }
    }
  ]
});

if (!user) { /* ... */ }

const isSubscribed = user.Subscriptions.length > 0;
```

Here, the fact that the `User` and `Subscription` models are associated is exploited to fetch the relevant subscription when the user is fetched (thanks to the `include` clause on lines 290-299). Then, whether or not there is a subscription turns into a simple offline check on line 304. In a small test, we found that the refactored code improves the performance of subscribing to a user from 99.2ms to 84.7ms (10 run times recorded pre- and post-refactoring, statistically significant difference at 95% confidence with a two-tailed homoscedastic T test). There are many refactoring opportunities like this in the `youtubeclone` application that was referenced throughout this chapter.

Chapter 6

Software Debloating

Abstract

JavaScript is an increasingly popular language for server-side development, thanks in part to the Node.js runtime environment and its vast ecosystem of modules. With the Node.js package manager `npm`, users are able to easily include external modules as dependencies in their projects. However, `npm` installs modules with *all* of their functionality, even if only a fraction is needed, which causes an undue increase in code size. Eliminating this unused functionality from distributions is desirable, but the sound analysis required to find unused code is difficult due to JavaScript's extreme dynamicity.

We present a fully automatic technique that identifies unused code by constructing static or dynamic call graphs from the application's tests, and replacing code deemed unreachable with either file- or function-level *stubs*. Due to JavaScript's highly dynamic nature, call graph construction may suffer from unsoundness, i.e., code identified as unused may in fact be reachable. To handle such cases, if a stub is called, it will fetch and execute the original code on-demand to preserve the application's behavior. The technique also provides an optional *guarded execution mode* to guard application against injection vulnerabilities in untested code that resulted from stub expansion.

This technique is implemented in an open source tool called *Stubbifier*, designed to help package developers to produce a minimal production distribution. *Stubbifier* supports the ECMAScript 2019 standard. In an empirical evaluation on 15 Node.js applications and 75 clients of these applications, *Stubbifier* reduced application size by 56% on average while incurring only minor performance overhead. The evaluation also shows that *Stubbifier*'s guarded execution mode is capable of preventing several known injection vulnerabilities that are manifested in stubbed-out code. Finally, *Stubbifier* can work alongside *bundlers*, popular JavaScript tools for bundling an application with its dependencies. For the considered subject applications, we measured an average size reduction of 37% in bundled distributions.

6.1 Introduction

JavaScript is one of the most popular programming languages, and has been the lingua franca of client-side web development for years [96, 200]. More recently, platforms such as Node.js [173] have made it possible to use JavaScript outside of the browser. Node.js provides a light-weight, fast, and scalable platform for writing network-based applications, enabling web developers to use the same language for both front- and back-end development. As a result, server-side JavaScript development has experienced an exponential growth in recent years.

This has given rise to a flourishing ecosystem of libraries, known as Node modules, that are freely available and widely used. The `npm` [170] package-management system in particular has fostered higher developer productivity and increased code reuse by unburdening the programmers from many routine development tasks. As such, a typical Node module m can directly and indirectly rely on myriad other modules. While an essential attribute of this ecosystem, in practice, m typically uses only a small fraction of the functionality of its dependencies, while still encompassing all of their code. In turn, clients of m inherit the unused functionality of m and its dependencies, as well as that of its own dependencies. The problem of accumulating code that in practice is never invoked is known as code "bloat".

While eliminating code bloat is desirable, "debloating" Node.js applications is challenging since it is nearly impossible to perform sound static analysis on JavaScript due to the high dynamism of the language. Despite the popularity of Node.js development and the severity of this issue, there is currently no technique available that can significantly debloat a modern Node.js application while fully preserving its original behavior.

Previous work on debloating JavaScript applications has been done in the context of JavaScript *bundlers* [185, 231]. The primary goal of bundlers is to create self-contained application distributions, but they typically perform an optimization known as "tree-shaking" [149] on imported external modules, by removing modules or functions that are unreachable in an application's import graph. Unfortunately, the size reduction achieved by bundlers is limited by the all-or-nothing nature of their code minimization technique: code that the bundler removes must *never* be called, else the bundled application will crash. Moreover, tree-shaking can only be applied to modern JavaScript code that uses the ECMAScript module system [149].

Another approach to debloating JavaScript applications was developed by Koishybayev and Kapravelos [126], who developed Mininode, a tool for reducing the size of *development distributions* of Node.js applications. In the JavaScript `npm` package ecosystem, a dis-

tinction is made between an application's dependencies and *development* dependencies: A dependency is another package that the application needs to function (e.g., a utility library such as `lodash`), whereas a development dependency *is only needed during development* (e.g., a test runner such as `mocha` that is needed to run the application's tests) and is not normally part of a production distribution. Mininode assumes an application's development distribution as the starting point and considers development dependencies and package tests as targets for removal. Further, Mininode completely removes code deemed unreachable through (unsound) static analysis and it only supports the ECMAScript 5 version of JavaScript (which dates back to 2009), which lacks modern JS features such as ES6 modules, classes, and async/await. Section 6.4.3 reports on an experiment in which we applied Mininode to the subject applications that we used to evaluate *Stubbifier*.

Previous work on debloating in the context of other languages has focused on the use of static analysis to determine unreachable code [37, 112, 174, 211]. In many existing techniques, the application stops executing when trying to invoke code that has been removed by the debloating algorithm and deviates from the intended behavior of the original application. Despite more recent advances for analyzing client-side web applications [138, 45, 115, 132, 133, 199], the development of a static analysis for Node.js that is simultaneously sound, precise, and scalable remains beyond the current state of the art.

This chapter presents a practical technique for reducing the size of production distributions of Node.js applications while preserving their original behavior. Core application functionality, as well as the extent to which an application uses its dependencies, is inferred automatically from dynamic or static call graphs constructed from the application's own test suites (which can be comprehensive, end-to-end test suites). Rather than using a sound call graph analysis and removing the code entirely, our approach relies on a fast, scalable, unsound call graph analysis, and untested, unreachable code is replaced by *stub* versions in a technique known as *code splitting*, pioneered by the Doloto tool [138]. If a function or file stub is executed, it will dynamically fetch and execute the original code so as to preserve application functionality. The technique has been implemented in a tool called *Stubbifier*, designed to be used by package developers looking to prepare a minimal production distribution for their package. *Stubbifier* improves on Doloto by: (i) supporting all features of modern JavaScript [74], including classes, promises, async/await, generators, and modules, (ii) introducing file-level stubs in addition to function-level stubs (so as to achieve additional debloating by stubbing all code in files where no code is used, instead of stubbing each of the functions in these files individually), and (iii) providing an (optional) *guarded execution mode*, where stubbed-out code is automatically instrumented to intercept calls to functions such as `eval` and `exec` that may introduce injection vulnerabilities. Most importantly, (iv) *Stubbifier* is *fully automatic* by relying on static analysis or

dynamic analysis of the application's test suite to identify code that is likely to be unreachable, whereas DOLOTO required traces of users interacting with the subject application to establish core application functionality.

Stubbifier was evaluated on 15 of the most popular Node.js applications, using five clients for each subject application to evaluate how much code is loaded dynamically. This evaluation found that *Stubbifier* achieves significant size reductions (56% on average), that the number of stubs expanded during the execution of client applications is relatively small, and that minimal performance overhead is incurred. Further, experiments with *Stubbifier*'s guarded execution mode confirmed that it is capable of preventing known injection vulnerabilities. Finally, we confirmed experimentally that, when used in conjunction with the popular `Rollup` bundler, *Stubbifier* achieves significant additional size reductions on previously bundled applications (37% on average).

In summary, this chapter describes:

- A fully automated technique for reducing the size of Node.js applications while preserving their original behavior, based on a combination of static or dynamic analysis and code splitting.
- The implementation of this technique in a tool called *Stubbifier* that supports modern JavaScript [74]. *Stubbifier* is publicly available as an open-source tool[1], and a self-contained code artifact including reproducible experiments is also available on Zenodo [217].
- An empirical evaluation of *Stubbifier* on 15 open source Node.js applications and 75 clients of these subject applications (five clients per subject), showing that *Stubbifier* reduces the size of Node.js applications by 56% on average while incurring only minor performance overhead. The evaluation also shows that *Stubbifier*'s guarded execution mode is capable of preventing several known injection vulnerabilities that are manifested in stubbed-out code.

The remainder of the chapter is organized as follows: § 6.2 discusses and/or directs readers to relevant background, and presents motivation for the work; § 6.3 discusses the code splitting approach in detail; § 6.4 presents an evaluation of the tool, and compares *Stubbifier* to a related tool called Mininode [126] in § 6.4.3; § 6.5 establishes some threats to validity; § 6.6 puts this chapter into context with respect to the related literature; § 6.7 concludes the presentation of code splitting; and finally § 6.8 relates this chapter to the others in the thesis.

[1] See https://github.com/emarteca/stubbifier.

6.2 Background and Motivation

The `npm` ecosystem includes more than 1.7 million modules[2] that provide a wealth of convenient features. By importing these libraries and reusing their functionality, programmers can focus their efforts on features that are unique to their application. However, this convenience does not come without its price: importing modules can cause projects to become excessively large due to the transitive importing of other projects that they depend on. In practice, it is often the case that a module only uses a small subset of the functions in the transitive closure of its dependencies.

To illustrate this, consider the example of a popular node application `css-loader` [233], a utility package for loading, parsing, and transforming `CSS` files and further supporting applications designed to use `CSS`. `css-loader` is one of the most popular modules on `npm`, with nearly 15 million weekly downloads, and it is imported by over 15,000 modules.

`css-loader` has 13 third-party production dependencies[3] (i.e., modules upon which its functionality depends). The stand-alone `css-loader` module contains only 16 files comprising 110KB. However, installing `css-loader` with direct and transitive production dependencies creates a package with 1299 files and total code size of 2764KB. This is a >81x and >25x increase in number of files and code size respectively.

To determine what part of the resulting installation constitutes application bloat, we examined `css-loader` to determine which functions and files are reachable from the application's test suite. Using a simple static analysis that traces function calls to build a list of unreachable functions and files, 209 files were found to be potentially unreachable, and 6 unreachable functions were identified in otherwise reachable files. Given the extreme dynamicity of JavaScript, sound static analysis is not possible [183, 199, 114, 142, 203]. Since in practice all static analyses for JavaScript suffer from unsoundness, some of the functions and files that they identify as being unreachable may indeed be reachable. Nevertheless, if one *could* devise a technique to remove all of this code, the application's size would be reduced by 80%.

Consider `semver` [171], a package that `css-loader` depends on. Only *two* functions from `semver` are used in `css-loader`: The `satisfies` function is imported specifically as part of the primary `css-loader` functionality, and the `inc` function is used once as a

[2] See `http://www.modulecounts.com/`.

[3] Many npm modules rely on additional development dependencies (sometimes referred to as "devDependencies") that are needed only for development purposes, e.g., for running tests. These dependencies are typically not installed by clients.

helper in the `css-loader` tests. Given that, it seems wasteful to include the *entire* `semver` code in `css-loader`.

In subsequent sections, we will describe our approach to addressing this issue of code bloat, and describe our implementation of this approach in a tool called *Stubbifier*. The intended user of our tool is a package developer looking to prepare a minimal production distribution for their package, e.g., a developer of `css-loader` might run *Stubbifier* on the package before a release. *Stubbifier* identifies the extent to which `css-loader`'s dependencies are used. For example, one of `css-loader`'s dependencies is `semver`: this developer would note that *Stubbifier* identifies 27 of `semver`'s files and six `semver` functions inside of `css-loader` to be potentially unreachable, i.e., `css-loader` imports `semver` but only uses a subset of imported functionality [4]. The six unreachable functions are in the file that exports the `inc` function. After debloating, the code in the `semver` package is reduced from 57KB to 35KB, a 38% size reduction. Overall, the size of `css-loader` as a whole is reduced by 80%, from 2.8MB to 0.6MB.

Note that the majority of the code removal is in the *dependencies* of `css-loader`. To illustrate, note that the initial size of `css-loader`, before installing any dependencies, is 110KB. The size of `css-loader` with all dependencies installed is 2.8MB, and *Stubbifier* reduces this to 0.6MB. This is 2.2MB of reduction, and since the original size of `css-loader` is just 110KB, *Stubbifier* must have mostly removed code in the dependencies of `css-loader`.

`css-loader` has approximately 14.8 million weekly downloads, so an 80% size reduction would translate to a reduction in weekly data transfer from 41.4TB to 8.8TB. We will elaborate on this in Section 6.4.

The next sections will present our debloating technique and its evaluation.

6.3 Approach

The debloating technique presented in this chapter involves several key steps, illustrated in the diagram in Figure 6.1. First, a call graph is computed for the project using its own tests as entry points, either dynamically by running the tests with instrumentation, or statically by running a static analysis. The project source code and call graph are input to the debloating algorithm: the technique essentially replaces functions and files that are *not* in the call graph with stubs, which are smaller but are equipped to fetch and load the

[4] There is more unreachable code in `css-loader`, but we focus on `semver` for the sake of illustration

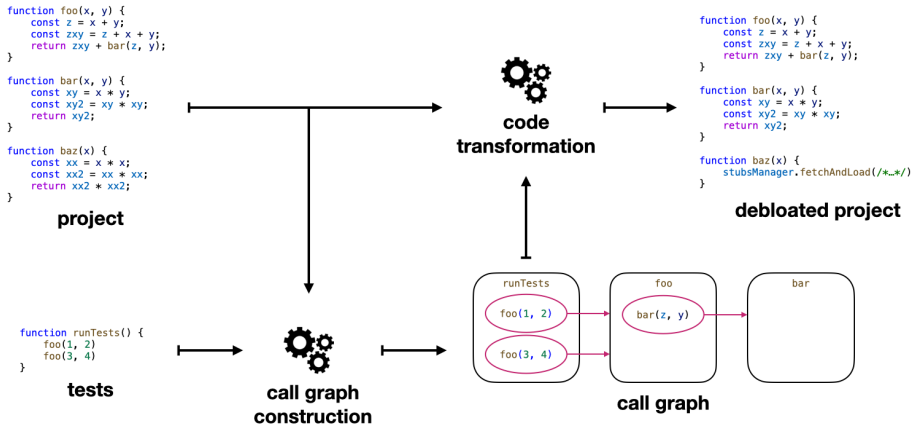


Figure 6.1: Overview of approach. A **call graph** is computed from a **project** using its **tests** as entry points. A **call graph construction** algorithm maps call sites to functions; e.g., here the `runTests` function contains two function calls `foo(1, 2)` and `foo(3, 4)`, both of which are mapped to the `foo` function, which contains a call `bar(z, y)` that is mapped to `bar`. Then, our debloating technique performs a **code transformation** to replace unreachable code (according to the call graph) with stubs that can dynamically load the code on-demand. Function `baz` is deemed unreachable since it does not appear in the call graph, and hence is replaced with a stub.

code dynamically if they are invoked. The end result is a debloated project that is ready to deploy.

We envision this technique to be used by developers that wish to create minimal distributions for their applications. The purpose of using an application's tests to infer unused functionality is to automatically determine the extent to which the application exercises its direct and transitive dependencies: If application tests had 100% coverage and the application fully exercised its dependencies, then no stubs would be introduced. In practice, however, a package will not use all of the code in its dependencies (e.g., `css-loader` includes `semver` but uses only a few of its functions). Note that the ideal scenario is when an application's tests have 100% *application* code coverage: in such cases, the unused parts of the application's *dependencies* would be replaced with stubs and nothing would ever be loaded dynamically.

The remainder of this section will discuss each step of the approach in detail.

6.3.1 Call Graph Construction

In principle, any call graph can be used to determine which files and functions should be replaced with stubs. The soundness and precision of the call graph will impact the size of the initial distribution and the amount of code that needs to be loaded dynamically.

The implementation of *Stubbifier* includes mechanisms for constructing a static or dynamic call graph. In each case, *Stubbifier* uses the test suite of the input application as the entry point for the analysis, and so the call graph represents the *tested* code. Any function that is not in the call graph is deemed unreachable and untested and will be replaced with a (*function-level* or *file-level*) stub. Both analyses are configured to consider depended-upon modules (in the `node_modules` directory), though note that development dependencies are excluded as they are typically not packaged and shipped with the subject application.

Below, we provide some further detail on the specific static and dynamic call graph construction techniques that *Stubbifier* supports.

Dynamic Call Graphs. To compute *dynamic* call graphs, code coverage is determined using Istanbul's command line tool `nyc` [113], that computes statement, line, branch, and function coverage for Node.js applications. By default, `nyc` ignores a project's dependencies, but *Stubbifier* automatically generates a configuration file that specifies that coverage of *non-development*, production dependencies should be computed. *Stubbifier* then runs `nyc` on the application's tests, to determine which functions and files are invoked during testing (and by exclusion, which were not invoked).

Static Call Graphs. To compute the *static* call graphs, we developed an analysis using CodeQL [51], GitHub's declarative language for static analysis, using its extensive libraries for writing static analyzers [97]. In particular, CodeQL's dataflow library contains functionality for tracking calls through local module imports, and we implemented an extension to recognize modules in a project's `node_modules` directory, and extended CodeQL's libraries to track data flow through these modules. Then, a call graph construction algorithm was implemented on top of this analysis, which uses the application's tests as entry points for the analysis.

This is an unsound analysis, as the use of dynamic features such as `eval` and dynamic property access expressions may give rise to missing edges in the call graph. We found that these dynamic features are so prevalent in modern JavaScript applications that using a sound, conservative call graph analysis is impractical (making conservative assumptions in the presence of these dynamic features would result in almost all code to be deemed

reachable). In our approach, reachable code mistakenly classified as unreachable due to the unsoundness of the analysis does not result in an error when called: rather, it is dynamically loaded via the stub.

6.3.2 Introducing Stubs

After constructing a call graph, *Stubbifier* creates lists of unreachable functions and files. Here, *unreachable files* are those in which *none* of the functions are reachable, and *unreachable functions* are those functions that are not reachable but that are in a file where at least one other function *is* reachable.

Next, *Stubbifier* parses the application's source code, including any dependencies, and replaces unreachable functions and files with *stubs* via transformations on the program's Abstract Syntax Tree (AST). Note that *Stubbifier* does not replace functions or files with stubs if they are shorter than the stubs that would replace them.

File Stubs. Each unreachable file is replaced with a *file stub*. The code in this stub implements Algorithm 1, which depicts the general logic for file stub expansion.

Algorithm 1: ExpandFileStub

```
1 perform all imports;
2 let file_o := fetchOriginalFileCode();
3 let file_e := eval(file_o);
4 replace this file with file_o;
5 perform all exports;
```

At a high level, file stub expansion amounts to: (i) performing all imports that were in the original code (line 1), (ii) fetching the original code and evaluating it (lines 2-3), (iii) replacing the contents of the stubbed file with the original file (line 4), and finally (iv) performing necessary exports (line 5). More specifically, in files that rely on the CommonJS mechanisms (i.e., `require` for importing and `module.exports` for exporting), simply storing the original code elsewhere and `eval`-ing it as needed suffices, as these mechanisms can be used anywhere in a source file. However, the ECMAScript Module System (ESM) [75]'s static `import`/`export` constructs cannot be executed in an `eval` (see ECMAScript 2019, section 15.2 [74]), so all `import` and `export` statements are hoisted out of the original code

```
305  // file.js before stubs are introduced
306  export function foo() { /* ... */ }
307  import { A };
308  function bar() { /* ... */ }
309  export default bar;
310
311
312  // file.js after stubs are introduced
313  import { A };
314  exportObj = eval(stubs.getCodeForFile("file.js"));
315
316  let foo_UID = exportObj["foo"];
317  export {foo_UID as foo};
318  export default exportObj["default"]
```

Figure 6.2: File before and after stubs are introduced.

and into the stub. The original code is then transformed to properly produce the values of the exports. To illustrate, consider the example in Figure 6.2.

In Figure 6.2, we see `import` and `export` statements interspersed through the file before stubs are introduced. In the lower part of the figure, we see that the file stub generated by *Stubbifier* contains all `import` statements as-is, and `export` statements are modified (lines 317 and 318) to get their values from the dynamically executed code (i.e., from `exportObj`, line 314).

To allow this exporting, the original code from Figure 6.2 is modified to construct an object containing all of the original exports. This constructed object is the last statement that will be executed when the code is passed to `eval`, and is therefore the return value of `eval`. This is illustrated in Figure 6.3.

```
319  function foo() { /* ... */ }
320  function bar() { /* ... */ }
321
322  {  foo: foo,
323     default: bar };
```

Figure 6.3: Modified original code with ES6 imports and exports (this is what would be `eval`'d).

Here, we see that the `export` was removed from the definition of `foo`, and that `foo` was added to an object on line 322, which also includes an entry for `bar`, the default export. The last statement in an `eval`-ed code block is implicitly returned—here, that is an object

containing the exports—allowing the stub to retrieve the exported values (as in line 314 of Figure 6.2).

Function Stubs. Functions deemed unreachable are replaced with *function stubs*. These stubs implement Algorithm 2, which depicts the general logic for dynamically loading and executing code upon stub expansion. When a stub is expanded, it first fetches the code,

Algorithm 2: ExpandFunctionStub

```
Data: args: function arguments
Data: uid: unique ID for this function stub
if uid cached then
    let fun_str := code cached at uid;
else
    let fun_str := fetch original function code;
let fun_e := eval(fun_str);
copy function properties to fun_e;
if can replace function definition then
    replace stub with fun_e;
else
    cache fun_str;
call fun_e with args;
return result;
```

either by retrieving it from a cache, fetching it from a server, or otherwise retrieving it from storage. Either way, the code is evaluated into a function, and function properties are copied from the stub version to the newly created function object. If possible, *Stubbifier* will replace the stub with the freshly evaluated original function (the conditions where this is or is not possible are discussed below). If not, the code is cached, and then the function is executed.

Stubbifier's caching strategy differs from DOLOTO's [138]: where DOLOTO caches function objects, *Stubbifier* caches the code, and we discuss the reasoning behind this shortly.

A concrete example of a function stub can be found in Figure 6.4, where we show the stub for `getValidHeaders` from the `node-blend` [22] project.

First, note that *Stubbifier* outfits each file with a global `stubs` object containing the code cache and functionality to fetch code. We see a call to `stub.getCode("UID_for_LOC")`

```
324  function getValidHeaders(headers) {
325    let toExec = eval(stubs.getCode("UID_for_LOC"));
326    stubs.cpFunProps(getValidHeaders, toExec);
327    getValidHeaders = toExec;
328    return toExec.apply(this, arguments);
329  }
```

Figure 6.4: Example of a stubbed function.

on line 325, which fetches the *original function definition* (found through `"UID_for_LOC"`, a unique ID for the function that *Stubbifier* generates from the code location when the stub is created). That code is then passed to `eval`, which will return a function object containing the original code. Line 326 copies any function properties from `getValidHeaders` to the fresh function[5]. Finally, line 327 redefines the `getValidHeaders` with the expanded stub, and line 328 calls the function with its original arguments[6]. Since `getValidHeaders` has reassigned itself on line 327, any subsequent calls to this function will call the expanded stub, with no need to re-`eval` the code.

The above discussion covered the general approach for introducing function stubs. However, several types of functions require special treatment, as will be discussed next.

Anonymous Functions. In JavaScript it is possible to create a function without a name, an idiom that is commonly seen when functions are passed as callback arguments to higher-order functions. In these cases, the function cannot reassign itself as is done on line 327 in the above example (since it has no name to refer to itself by), so the loaded code is cached, and future stub expansions `eval` the cached code. For example, Figure 6.5 displays the `getValidHeaders` stub that we would create if this function did not have a name. Here, rather than immediately passing the code loaded with `stubs.getCode("UID_for_LOC")` to `eval`, the `stubs` cache is accessed on line 331. Code is only loaded on a cache miss, in which case the loaded code is immediately cached.

One might wonder why the function stub expansion caches the loaded code, evaluating it every time the stub is invoked, rather than caching the expanded function object. This is necessary because, in JavaScript, functions are *closures* that *close* variables from surrounding scopes directly into the object. Therefore, generating a stub for a function that

[5]Recall that in JavaScript functions are objects, and can have properties assigned dynamically.

[6]`apply` calls its receiver as a function, binding its first argument to `this` inside the function, and passing the other arguments as function arguments. `arguments` is a metavariable available inside functions that refers to its arguments.

```
function(headers) {
    let toExecString = stubs.getStub("UID_for_LOC");
    if (! toExecString) {
        toExecString = stubs.getCode("UID_for_LOC");
        stubs.setStub("UID_for_LOC", toExecString);
    }
    let toExec = eval(toExecString);
    toExec = stubs.cpFunProps(this, toExec);
    return toExec.apply(this, arguments);
}
```

Figure 6.5: Example of stubbed anonymous function.

is nested inside another would include the *function arguments* of the latter in its closure. If we were to cache this object, any subsequent call to the function would refer to the values of function arguments when the stub was first expanded, which may lead to incorrect program behavior. Thus, we have to `eval` every time. Note that this problem does not arise for functions with a name, as the function reassigning itself does not store a closure.

DOLOTO cached function closures, which is problematic for the reasons discussed above; we conjecture that the authors did not evaluate their tool on code where this issue would arise.

Class and Object Methods. When replacing object or class methods with stubs, an issue arises that relates to references to `this`. In functions outside a class or object, `this` refers to the function object itself, while in a class/object, `this` refers to the *object instance* on which the function was invoked. These class methods need to be referenced in a different way to allow for function property copying and reassignment.

Fortunately, class and object methods can be accessed as *properties* of `this`, and so if `getValidHeaders` were a method in a class, the following replacements would be made:

```
// outside a class/object
stubs.cpFunProps(getValidHeaders, toExec);
getValidHeaders = toExec;

// inside a class/object
stubs.cpFunProps(this.getValidHeaders, toExec);
this.getValidHeaders = toExec;
```

For class/object methods with no ID, we generate a dynamic property access on `this` to reassign the function object as at code generation time, we know the key corresponding to nameless object properties. Specifically, this means that instead of `this.functionName` we use `this[${generate(key)}$]`, where `${generate(key)}$` is a string generated at parsing runtime, to reference the function as a dynamic property access on `this`.

Classes and objects can also have *getter* and *setter* methods, as is illustrated in the example below:

```
347  class A {
348      get propName() { console.log("getter"); }
349      set propName() { console.log("setter"); }
350  }
351  let x = new A();
352  x.a; // prints "getter"
353  x.a = 5; // prints "setter"
```

Getter and setter stubs are generated with special reassignment code. Dynamically accessing and defining a getter for some property `"p"` is done using `this.__lookupGetter__("p")` and `this.__defineGetter__("p")` respectively (and similarly for setters); these calls are used in place of direct accesses as properties of `this` in the stub.

Arrow Functions. Arrow functions were introduced in ECMAScript 2015, and provide a more concise syntax for functions. When creating stubs for arrow functions, we run into an issue as the metavariable `arguments` cannot be used to reference the function arguments. To get around this, we make use of the *rest parameter* [161], also introduced with ES6. By replacing the original function parameters with a rest parameter, we have essentially recreated the functionality of `arguments`. For example, if `getValidHeaders` were an arrow function, it would be written as:

```
354  let getValidHeaders = (headers) => { /* elided function body */ }
```

and its stub would resemble:

```
355  let getValidHeaders = (...args_UID) => {
356      // only change the last line of the stub
357      getValidHeaders.apply(this, args_UID);
358  }
```

Unstubbable Functions. *Stubbifier* does not transform generators, as `yield` cannot be present inside of an `eval`, nor does it transform constructors. Constructors necessitate that `super` be called before any use of the `this` keyword. Generating constructor stubs would require a more sophisticated analysis of constructor code, and as sound static analysis of JavaScript is still very challenging, we decided against stubbifying them altogether. We do not consider this to be a big issue, as these types of functions are fairly rare; we only encountered a few instances of unreachable constructors or generators in our evaluation.

Manually specifying functions *not* to stub. Users may be interested in specifying some functions that should never be replaced with stubs, regardless of their classification in the generated call graph. To accommodate this, we added functionality to allow users to manually flag a function so it will be ignored by *Stubbifier*. To illustrate the usefulness of this feature, consider a developer that has been using *Stubbifier* for some time. This developer may note that *Stubbifier* classified some function as unreachable, but that function is nearly always loaded dynamically in clients of the developer's package. Instead of writing tests for this function (e.g., perhaps the function is difficult to write unit tests for), the developer can configure *Stubbifier* to ignore that function to avoid it needing to be loaded dynamically.

6.3.3 Guarded Execution Mode

Since *Stubbifier* builds the input call graph using the application's tests, the stubbed code is also the *untested* code. Dynamically loading and executing this code could pose a security risk, as it may include injection vulnerabilities that were not encountered during testing.

To address such concerns, *Stubbifier* includes an option to detect calls to a pre-specified list of "dangerous" functions in expanded code. This is achieved by intercepting all function calls and checking whether or not the function is (perhaps an alias of) one of these dangerous functions. In our current implementation, the list of these functions consists of: `eval`, `process.exec`, and `child_process.{fork, exec, execSync, spawn}`, common functions that enable the execution of arbitrary code. It is trivial to include other functions to this list, so users can customize what functions they want guard against. We include an example of the code with guards in Appendix C.

These checks can be configured to generate a warning, or exit the application if a dangerous function is about to be called. This transformation is run on the original code so that, when a stub is expanded, the loaded code includes guards.

Since these functions could be aliased, we must wrap *every* function call with these checks. As such, the size of the loaded code (i.e., the expanded stubs) is increased dramatically. The guards also incur more runtime overhead, as will be discuss in Section 6.4.

6.3.4 Asynchrony

JavaScript is a single-threaded language, and so our approach need not deal with the multi-threaded setting. That said, JavaScript does provide several mechanisms for asynchronous

programming (event-driven programming, promises and async/await), and the use of these features may give rise to imprecision and unsoundness during call graph construction. Our approach addresses this by not assuming soundness in the first place–stubs are introduced in cases where the analysis has identified a function as being unreachable. Unsoundness will cause more stubs to be introduced, which increases runtime overhead when they are expanded.

6.3.5 Bundler Integration

Many JavaScript projects use *bundlers* such as `webpack` [231] and `Rollup` [185] to package an application along with all the modules that it depends on into a single-file distribution that includes all required functionality. Such a bundle can be included in another application using `require` or `import`, so that users do not need to go through additional installation steps.

Bundlers perform a limited form of code debloating known as "tree-shaking", which identifies functions and classes that are unused based on a static analysis of the import relationships between modules. If the project relies on `require` statements to import external functionality, the required files are simply included in the bundle in their entirety; if the project relies on the ECMAScript Module System, parts of an imported module can be removed if they are not referenced in the importing module. The size reductions that can be achieved using tree-shaking can be significant, but they are still limited by the fact that soundness is required, because the removal of code that is used could cause a bundled application to crash.

The use of *Stubbifier* in combination with a bundler requires a few additional steps in the previously discussed transformation pipeline. First, bundling must always happen *before* applying *Stubbifier*, as bundlers perform their own code transformations. For example, when merging all the application code into a single file, bundlers often refactor the code so as to avoid variable name conflicts, repeated imports, etc. If *Stubbifier* were run on the application before bundling, the bundler would only perform its analysis on the code that is not replaced with stubs, since the code to be loaded dynamically is just stored as plain text. As a result, expanding a stub would result in code that does not match, e.g., the changed variable names in the bundle, which is likely to result in errors.

To prevent such issues, *Stubbifier* should be applied to an application *after* it has been processed by a bundler. One minor obstacle here is that *Stubbifier* uses an application's tests as the entry points for call graph construction, and tests are nearly always based on the original project source code, and not on a bundle. To address this, *Stubbifier*

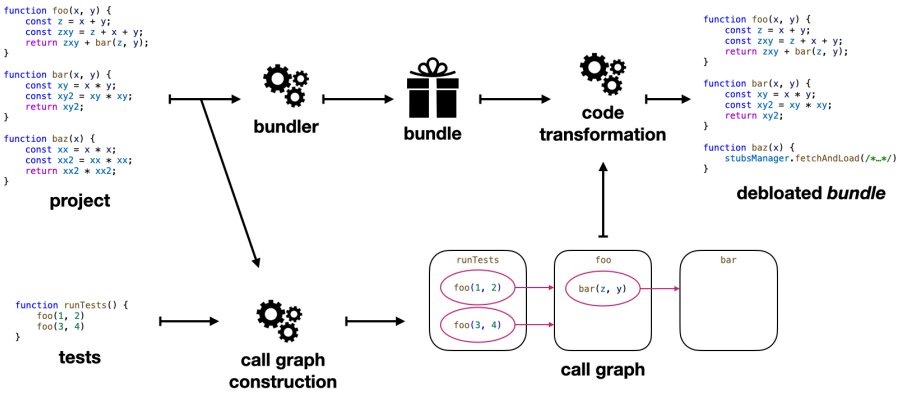


Figure 6.6: *Stubbifier* overview with bundler integration. As in Figure 6.1, a call graph is computed from the project and its test suite, but here the project is bundled before being transformed. The code transformation is applied to the bundle w.r.t. the call graph computed from the application (note: test suites typically rely on the non-bundled application, which is why the call graph is not computed over the bundle directly).

determines a mapping of the functions to be stubbed from their positions in the original code to their positions in the bundle. Then, it constructs call graphs from tests as discussed before, and it consults the mapping to determine where stubs should be introduced in the bundle. This is all illustrated in the diagram in Figure 6.6, which is expanded from the diagram in Figure 6.1. The major difference with the approach illustrated in Figure 6.1 is the use of a bundler *after* the call graph is computed, but *before* debloating; the result of this entire process is a debloated application bundle.

The evaluation presented in Section 6.4 will examine how much additional code size reduction can be achieved by *Stubbifier* on applications after they have been bundled using `Rollup`.

6.4 Evaluation and Discussion

This section presents an evaluation of *Stubbifier* that aims to answer the following research questions:

- **RQ1.** How much does *Stubbifier* reduce application size, and which type of call graphs (static or dynamic) is more effective for reducing application size?
- **RQ2.** How much code is dynamically loaded due to stub expansion?

- **RQ3.** How much overhead is incurred due to stub expansion?
- **RQ4.** How much time does *Stubbifier* need to transform applications?
- **RQ5.** How much run-time overhead is incurred by guarded execution mode and can it detect security vulnerabilities?
- **RQ6.** How much does *Stubbifier* reduce the size of applications that have been bundled using `Rollup`?

6.4.1 Experimental Setup and Methodology

To evaluate *Stubbifier*, we selected 15 projects from the most popular projects published by `npm`; specifically, we listed projects in descending order by number of weekly downloads, and went down this list, selecting a project if it met the following two criteria: first, we required that the project installed, was able to build without error, and had a running test suite with no failing tests (as *Stubbifier* uses the test suite to generate call graphs). If a project satisfied these criteria, we then randomly selected from its *dependents*, or *clients*, and attempted to install, build, and run their tests; if the project had five such clients, it was selected for our evaluation. The selection criterion that was the most difficult to satisfy is that subject applications needed to have at least 5 client packages that had fully passing test suites. The availability of such client packages is critical to our evaluation since this provides us with a way to assess frequency and cost of stub expansion in a realistic setting (since we introduce stubs in an application based on its own tests, running the same tests to evaluate stub expansion would have yielded biased results).

The intended user of *Stubbifier* is a package developer: when the developer is ready to prepare a production distribution of their package, they can run *Stubbifier*. Based on the application's tests, *Stubbifier* will determine the extent of the package code that is reachable, *as well as the extent that the package exercises its dependencies*. Thus, it is likely that *Stubbifier* will remove large swathes of the package's production dependencies. The developer is left with a minimal distribution that they should feel safe distributing to users. The evaluation described here is intended to simulate that experience: we run *Stubbifier* on a package, and then insert the stubbified version of that package in five of the package's dependents to confirm that the debloated distribution works, and evaluate the extent to which our technique was effective (by running the tests of the dependents).

Project (citation)	Commit	LOC	# files	# tests	Test Coverage Src	Test Coverage Deps	Deps	Size (KB)
`memfs` ([24])	`a9d2242`	18k	133	284	80.7%	37.2%	1	146
`fs-nextra` ([4])	`6565c81`	11k	184	138	99.0%	99.0%	0	52
`body-parser` ([13])	`480b1cf`	20k	210	231	99.7%	29.6%	21	364
`commander` ([27])	`327a3dd`	13k	177	351	48.8%	48.8%	0	70
`memory-fs` ([29])	`3daa18e`	14k	167	44	97.4%	58.9%	11	120
`glob` ([19])	`f5a57d3`	13k	175	1706	95.9%	72.0%	10	86
`redux` ([182])	`b5d07e0`	105k	4491	82	96.9%	0.5%	2	267
`css-loader` ([28])	`dcce860`	71k	1299	430	99.3%	4.88%	36	2764
`q` ([21])	`6bc7f52`	16k	135	243	42.9%	14.9%	0	281
`send` ([23])	`de073ed`	14k	157	152	100%	68.5%	17	97
`serve-favicon` ([16])	`15fe5e3`	10k	121	30	100%	58.8%	5	20
`morgan` ([15])	`19a6aa5`	14k	159	81	100%	73.6%	8	55
`serve-static` ([17])	`94feedb`	13k	160	90	100%	48.4%	19	106
`prop-types` ([18])	`d62a775`	15k	152	287	98.0%	1.48%	4	106
`compression` ([14])	`3fea81d`	13k	149	38	100%	40.6%	11	66

Table 6.1: Summary of projects used for evaluation

Table 6.1 lists the projects used for the evaluation, as well as some relevant metrics. The first row reads: the project `memfs` has 18k lines of code (LOC) in the analyzed files[7], and there are 133 files analyzed (Num files). The `memfs` test suite has 284 tests, which have a coverage of 80.7% of the source code of the project (Coverage: Src), and a coverage of 37.23% of its production dependencies (Coverage: Deps). `memfs` has one production dependency (Deps), and its analyzed code comprises 146 KB (Size). Note that the number of dependencies includes both direct and transitive dependencies.

We have created a code artifact [217] to accompany this chapter: the artifact includes each project cloned at the version on which we ran the evaluation, the experimental infrastructure used to conduct said evaluation, as well as the full source code of *Stubbifier*.

Selecting subject applications. Each subject application was processed twice with *Stubbifier*, once using static call graphs and once using dynamic call graphs. In each case, files and functions deemed unreachable were replaced with stubs. To address **RQ4**, the time required for the entire process was measured. For **RQ1**, the size of the application before and after introducing stubs was compared. We compute the size of source code (excluding tests), *including* production dependencies and *excluding* development dependencies.

To address **RQ2** and **RQ3**, we selected five clients of each subject package from its list of dependents that is published on `npm`. These clients were essentially selected randomly, but we excluded clients without tests or with failing tests. We also confirmed that the

[7]The metrics in the table reflect the project's own source code (excluding tests), and all its (transitive) production dependencies, but excluding `devDependencies`. .

Project	Size (KB)	Reduction %	Expanded (KB)	Red after exp (%)
memfs	19	87%	[19, 138]	[87%, 5%]
fs-nextra	31	39%	[31, 45]	[39%, 14%]
body-parser	65	82%	[211, 297]	[42%, 18%]
commander	68	2%	[68, 68]	[2%, 2%]
memory-fs	41	66%	[41, 87]	[66%, 27%]
glob	61	28%	[70, 80]	[18%, 7%]
redux	201	25%	[221, 221]	[17%, 17%]
css-loader	559	80%	[559, 895]	[80%, 68%]
q	37	87%	[37, 100]	[87%, 64%]
send	59	39%	[59, 92]	[39%, 5%]
serve-favicon	15	24%	[15, 18]	[24%, 8%]
morgan	25	55%	[41, 45]	[25%, 20%]
serve-static	38	64%	[38, 83]	[64%, 21%]
prop-types	18	83%	[56, 56]	[48%, 48%]
compression	24	63%	[24, 24]	[63%, 63%]

(a) Size of projects stubbified with static CG

Project	Size (KB)	Reduction %	Expanded (KB)	Red after exp (%)
memfs	17	89%	[17, 136]	[89%, 7%]
fs-nextra	47	10%	[47, 47]	[10%, 10%]
body-parser	173	53%	[180, 253]	[51%, 31%]
commander	59	16%	[59, 59]	[16%, 16%]
memory-fs	100	17%	[100, 117]	[17%, 3%]
glob	84	4%	[91, 91]	[-6%, -6%]
redux	189	29%	[209, 209]	[22%, 22%]
css-loader	584	79%	[584, 1372]	[79%, 50%]
q	206	27%	[206, 209]	[27%, 26%]
send	89	8%	[89, 93]	[8%, 5%]
serve-favicon	19	3%	[19,19]	[3%, 3%]
morgan	49	13%	[52, 52]	[7%, 7%]
serve-static	98	7%	[98, 102]	[7%, 4%]
prop-types	16	85%	[53, 53]	[50%, 50%]
compression	46	29%	[46, 46]	[29%, 29%]

(b) Size of projects stubbified with dynamic CG

Table 6.2

dependency is actually used in the client: there are some projects that list a package as a dependency but no longer use in the source code, and we excluded these. Finally, we exclude clients that require the use of older versions of Node.js. We did not run an application's own test suite to explore **RQ2** and **RQ3** because it was debloated based on those very same tests. Further, the use case envisioned is that of a developer debloating their own project, and clients importing the debloated version; examining how the debloated version of a project behaves in the setting of one of its clients replicates this.

Conducting performance measurements. To determine the performance overhead caused by stub expansion, we compared the runtime of each of these clients' tests when using the stubbed and original subject application. When running the test suite with the stubbed application, we also tracked the total size and number of stub expansions to determine *how much* code is loaded dynamically. In our evaluation, stubs were loaded from local storage on the machine running the evaluation.

To mitigate noise and bias caused by caching, all test suites were executed 10 times after two test runs before the timed experiments; the reported results are the average of these 10 test runs. Furthermore, since some of the tests generate files in `/tmp`, this directory is cleared between every test suite run.

Finally, to mitigate versioning errors, we run our experiments on a client using the same version of the dependency as the one that we transform. Specifically, we do the following when testing a client:

- `npm` or `yarn install` in the root of the client project.
- Replace the dependency in question in the client's `node_modules` with a *symbolic link* to the source code of the dependency that we will transform.
- Run the client's tests.
- Transform the dependency. The symbolic link means the client needs no change to use the stubbed version of the dependency.
- Rerun the client's tests, now with the stubbed version of the dependency.

All our experiments were conducted on a Thinkpad P43s with an Intel Core i7 processor and 32GB RAM, running Arch linux, using the same version of Node.js (14.3.0), to avoid any updates to the runtime environment that could affect run times and thus skew the results.

Guarded execution mode. For **RQ5**, we repeated the experiments with guards enabled, and measured the running time and size of expanded code for the client test suites to determine the increase in overhead due to these extra checks.

In addition, we report on a case study involving `depd` [10], a subject application with a known vulnerability, and on experiments with `osenv` and `node-os-uptime`, two `npm` modules with confirmed vulnerabilities that were used as experimental subjects in Karim et al. [121][8].

Bundlers. For **RQ6**, each subject application was bundled use the `Rollup` bundler [185]. This involved the creation of a bundler configuration file (which we generated automatically given the application's `package.json` file) to bundle the application based on its listed entry points and to create a single bundle that also includes all of its production dependencies[9]. We measure and report on the sizes of the resulting bundle, both with and without having applied *Stubbifier*, to determine what additional size reduction is enabled by *Stubbifier*.

6.4.2 Overview of Results

The results of running *Stubbifier* on the projects are displayed in Tables 6.2a and 6.2b. We show the size of the original source code, the size of the application distribution, and then the resulting size of the distribution after we run our transformation on it, with both the static and dynamic call graphs.

Note that the size immediately after transformation is only representative of the stubbed application size if no stubs are expanded. To gain a realistic estimate of the size reduction in a standard use-case of the application, we identified five clients for each application and tracked how many stubs were expanded during the execution of the test suites of these clients. Then, we consider the size of the application to be its base stubbed size *plus* the total size of the stubs that were expanded during the client tests. This is reported as a range of the lower and upper bounds of application size over the five clients. The full data is included in Appendix C.

The first row of Table 6.2a reads: after running *Stubbifier* with the static call graph, the size of the `memfs` source code is reduced to 19KB, which is a reduction of 87% of the original

[8]Of the subject applications reported on in this work [121], these were the only two that had a confirmed vulnerability and a test suite with passing tests.

[9]The default behavior of `rollup` is to ignore dependent modules in `node_modules`, but the bundle should all code in which stubs may be introduced, to be able to determine *Stubbifier*'s effectiveness.

application size. This expanded to a minimum of 19KB (i.e., nothing was expanded) and a maximum of 138KB over the five clients tested; the expanded code is a reduction of 87% (with minimum expansion) and 5% (with maximum expansion) of the original application size. The first row of Table 6.2b can be read the same way, but for results after running *Stubbifier* with the dynamic call graph on `memfs` and testing with the same five clients.

In the remainder of this section, we will address each research question in order.

RQ1: How much does *Stubbifier* reduce application size, and which type of call graphs (static or dynamic) produces smaller applications?

We refer the reader to Tables 6.2a and 6.2b. In these tables, it can be seen that, using static call graphs, size reductions ranging from 2% to 87% are achieved (56% on average). The case where a size reduction of only 2% is achieved is `commander`, which has no dependencies and appears to be a bit of an outlier. Using dynamic call graphs, size reductions ranging from 3% to 89% achieved (31% on average).

Overall, the use of static call graphs results in larger size reductions in 11/15 cases, and in larger size reductions on average (56% on average when static call graphs are used vs. 31% when dynamic call graphs are used). This is not surprising, as both static and dynamic call graph constructions use the test suite as the entry point of the application, and the static analysis suffers from unsoundness due to the dynamic nature of JavaScript. Since the static analysis is constructing a call graph, unsoundness might cause some functions to be excluded from the call graph when they are actually executed in the test suite. As a result, the initial code size reduction is therefore usually larger, but more stubs need to be expanded at run time. The dynamic analysis finds every function that is called during the test suite execution, since it is constructed with a coverage tool. If the static analysis was perfectly precise then it would produce the exact same call graph as the dynamic analysis.

Many of these packages have millions of weekly downloads, and so the size savings add up quickly: for example, `css-loader` is 2.764MB, and with 10 million weekly downloads we have nearly 28TB of data transferred to users every week. *Stubbifier* reduces `css-loader`'s initial size by 80% with both call graphs, which would contribute to 22 fewer TB being transferred weekly (for one project!).

On average, *Stubbifier* reduces initial application size by 56% when using static call graphs, and by 31% when using dynamic call graphs.

RQ2: How much code is dynamically loaded due to stub expansion?

Again referring to Table 6.2a and 6.2b, this time to the **Expanded KB** range columns, we see that the top end of the expanded ranges using the static call graph are smaller than (or equal to) the expanded ranges using the dynamic call graph in 11/15 cases. This aligns with our findings in **RQ1**. In all but one case, the minimal expanded size is close to the reduced application size, and the maximum size increase is > 2x in only two cases.

The case where `glob` is processed using a dynamic call graph is an interesting outlier, as its size is *larger* than the original code after all stubs have been expanded. This is because not much of `glob` is stubbed (the initial size reduction is only 4%, or 2KB), and the code required to support stub expansion is larger than the initial size reduction due to the extra boilerplate that was introduced by *Stubbifier* (import statements, `eval` call, reassignments to imports, etc.).

To break down the results further, we consider the results for all clients of a few packages. Tables 6.3a and 6.3b display all the metrics tracked for all clients of `redux`, `q`, and `body-parser`. These metrics are the test suite runtimes, the percentage slowdown due to running the stubbed code, and number and size of stubs dynamically expanded during the tests. We chose these applications to display as we felt they are a representative sample of our results; the full data for all clients of all projects is included in Appendix C.

The first row of Table 6.3a reads: for `redux`, its client application `Choices` has an average test suite runtime of 5.06 seconds. When the `Choices` test suite is rerun with stubbed `redux` (via the static call graph), it has an average runtime of 5.16 seconds, which is a slowdown of 2%; 1 file stub and no function stubs were expanded, and the total size of stubs expanded was 20.06KB. The first row of Table 6.3b shows the results of rerunning again with stubbed `redux` via the dynamic call graph: now, `Choices`' test suite has an average runtime of 5.05 seconds, which is a slowdown of 0%; 1 file stub and no function stubs were expanded, and the total size of stubs expanded was 20.06KB.

Digging into the client-specific data reveals some interesting trends. There appears to be a correlation between the number of stubs expanded for the static and dynamic call graphs. For example, consider the clients of `body-parser`: even though there are more stub expansions using the static call graph vs. using the dynamic call graph, it appears that there are "sets" of functionality that are commonly expanded together (seen here as whenever 48 file stubs are expanded in the static case, 14 file stubs are expanded in the dynamic case). The range of expansions among clients suggest that some of the clients use more of an application's untested functionality than others.

	Client		Stubbed code: effect of expansions				
Proj	**Client Proj**	**Time (s)**	**Time (s)**	**Slowdown (%)**	**Files**	**Fcts**	**Exp (KB)**
redux	Choices	5.06	5.16	2%	1	0	20.06
	found	30.61	31.83	4%	1	0	20.06
	Griddle	8.93	8.91	0%	1	0	20.06
	react-beautiful-dnd	61.70	63.49	3%	2	0	20.06
	redux-ignore	0.57	0.58	2%	1	0	20.06
q	decompress-zip	0.70	0.74	6%	1	0	63.25
	downshift	1.43	1.44	1%	1	0	63.25
	node-ping	3.80	4.20	10%	1	0	63.25
	passport-saml	0.41	0.44	6%	0	0	0.00
	requestify	2.92	2.99	2%	1	0	63.25
body-parser	appium-base-driver	8.66	10.04	14%	39	0	146.10
	express	1.05	1.89	45%	48	0	231.69
	karma	2.08	2.12	2%	40	0	199.57
	moleculer-web	5.80	6.46	10%	48	0	231.69
	typescript-rest	13.17	14.89	12%	48	0	231.69

(a) Stubbed with static call graph

	Client		Stubbed code: effect of expansions				
Proj	**Client Proj**	**Time (s)**	**Time (s)**	**Slowdown (%)**	**Files**	**Fcts**	**Exp (KB)**
redux	Choices	5.06	5.05	0%	1	0	20.06
	found	30.61	31.34	2%	1	0	20.06
	Griddle	8.93	9.03	1%	1	0	20.06
	react-beautiful-dnd	61.70	62.12	1%	2	0	20.06
	redux-ignore	0.57	0.59	3%	1	0	20.06
q	decompress-zip	0.70	0.78	10%	0	5	2.98
	downshift	1.43	1.44	1%	0	1	0.88
	node-ping	3.80	4.08	7%	0	6	2.86
	passport-saml	0.41	0.42	2%	0	0	0.00
	requestify	2.92	3.05	4%	0	2	0.86
body-parser	appium-base-driver	8.66	9.26	6%	8	0	6.69
	express	1.05	1.21	14%	14	0	79.70
	karma	2.08	2.09	1%	12	0	79.03
	moleculer-web	5.80	6.38	9%	14	0	79.70
	typescript-rest	13.17	14.48	9%	14	0	79.70

(b) Stubbed with dynamic call graph

Table 6.3: Results for Clients of Select Projects

We also noted consistency in *which* stubs are expanded. For example, in the "sets" of expanded functionality described earlier, these are the *same* 48 and 14 files every time. As an additional example, all the clients of `redux` expand one file stub (one client expands two)—this is always the *same* stub that is expanded. In the other applications, there is always significant overlap in which stubs are expanded with different clients. This suggests that some of these applications have commonly used functionalities that are untested, so developers could use this information to shore up their test suites.

Finally, we observe that the dynamic call graph typically produces far fewer file stub expansions than the static call graph. There are a few dimensions to this. On one hand, as JavaScript is a dynamic language, the static call graph is likely to be incomplete—functions in JavaScript are often called in highly dynamic ways, and these kinds of calls are more easily detected using dynamic analyses. On the other hand, the dynamic call graph is more susceptible to lower-quality tests: if the application is poorly tested, the dynamic call graph will report many unreachable functions and files. It is not immediately clear which call graph yields "better" results, as fewer stubs mean less size reduction, but also less overhead—we ultimately leave the decision up to the developer.

Most package clients load very little code dynamically. Many applications have commonly loaded "sets" of code, representing broadly used, untested functionality.

RQ3: How much overhead is incurred due to stub expansion?

To determine the performance overhead introduced by stub expansion, we measured the running times of the test suites of clients of applications processed by *Stubbifier*.

We decided not to aggregate runtime information over all clients of a package as the overhead depends on many factors outside of our control: the number of tests, the structure of the tests, the raw running time of the test, etc. Instead, we conducted a case study on the effect of the dynamic code loading for the individual clients of the three projects presented in Tables 6.3a and 6.3b. The results for all test applications are included in Appendix C, but the trends are upheld across the full data.

Referring to the time columns of Tables 6.3a and 6.3b, the following conclusions can be drawn. First, a correlation between the slowdown and the number of stub expansions can be observed: as more code is dynamically loaded, the performance overhead increases. This aligns with our expectations, as stub expansions involve additional I/O and compute time. That said, the runtime overhead is never extreme, and the slowdowns still leave the running times of the test suites well within the same order of magnitude. As a percentage,

	Static CG		Dynamic CG	
Project	**CG generation (s)**	**Transf. (s)**	**CG generation (s)**	**Transf. (s)**
memfs	740.18	2.46	15.97	2.72
fs-nextra	380.97	1.11	13.94	1.10
body-parser	295.38	3.43	10.53	3.79
commander	554.93	1.94	24.56	1.61
memory-fs	324.06	1.73	5.33	1.75
glob	300.80	2.09	18.66	1.46
redux	1349.09	3.18	182.02	4.02
css-loader	1137.77	14.85	48.61	15.52
q	336.31	4.41	10.98	4.85
send	279.16	1.75	7.57	1.67
serve-favicon	259.06	0.76	3.91	0.79
morgan	313.76	1.20	8.65	1.16
serve-static	276.89	1.67	7.51	1.60
prop-types	752.94	2.12	12.79	1.89
compression	279.18	1.28	6.78	1.37

Table 6.4: Callgraph generation and transformation timing

some runtime overhead is high (e.g., `body-parser`'s `express` dependency), but the magnitude of the change is not (only 0.84 seconds). We do not see high percentage slowdowns for long-running tests, for instance `redux`'s `found` and `react-beautiful-ignore` clients have 4% and 3% slowdowns respectively. We conjecture that the amount of overhead mostly has to do with the I/O required to load the dynamic code.

By and large, the magnitude and percentage overhead introduced by dynamic loading is small.

RQ4: How much time does *Stubbifier* need to transform applications?

Table 6.4 shows the time needed by *Stubbifier* to process each of the 15 projects. Here, we distinguish between the time needed to construct call graphs, and the time needed to transform the source code. Note that as the execution of the project test suite is a necessary step for constructing the dynamic callgraph, the dynamic callgraph generation time includes the time required to run the tests.

The first row of the table reads: for `memfs`, generating the static call graph takes 740.18 seconds and applying transformations based on this call graph takes 2.46 seconds. Furthermore, generating the dynamic call graph takes 15.97 seconds and applying transformations based on this call graph takes 2.72 seconds.

From the table, it can be seen that the cost of the code transformation itself is negligible. The longest runtime is 15 seconds on the `css-loader` project, which is unsurprising given that `css-loader` is the largest subject application (2.76MB). There is no difference between the transformation times using the static vs dynamic call graphs. This is also unsurprising, as the same process is used to run the transformation in either case, and, generally, a similar number of stubs is created. In cases such as `q`, where the dynamic call graph produces a larger stubbed application and yet it takes longer to run, this is because there are more *function* stubs being generated (compared to a single file stub being generated when using static call graphs).

The cost of call graph construction is more noteworthy. Overall, we see that constructing a static call graph takes one to two orders of magnitude more time than constructing the dynamic call graph. We also observe a correlation between the times to construct the static and dynamic call graphs. To construct the dynamic call graph, *Stubbifier* simply computes a coverage report from running an application's tests (including `node_modules`), which amounts to the time to run the tests plus some small overhead. The slower runtime of static call graph construction is due to our inclusion of the generation of the CodeQL database in the overall runtime, which is directly proportional to the amount of code in the project (in order to run any static analysis queries, CodeQL must build a database of the application's code—this is a one-time cost as long as the code does not change).

We envision the use-case of *Stubbifier* to be a final stage in the creation of a production release, and so do not believe a build-time of 5-15 minutes to be prohibitive. If a user wanted to apply *Stubbifier* more frequently, they could opt for using dynamic call graphs.

The average runtime of *Stubbifier* with the static call graph is not prohibitive (at roughly 8.3 minutes), and is much lower (28 seconds) with the dynamic call graph.

RQ5: How much run-time overhead is incurred by guarded execution mode and can it detect security vulnerabilities?

The use of "dangerous" functions such as `eval` and `exec` that interpret string values as code is known to cause injection vulnerabilities in JavaScript applications [121]. It is particularly concerning if such functions are invoked from untested code, because it means that the developers may not have considered all situations where calls to such functions are executed. *Stubbifier*'s guarded execution mode aims to mitigate this risk, by adding dynamic checks for such functions in stubbed-out code so that a warning can be issued or execution can be terminated when such calls are encountered. These dynamic checks may

Client Proj	With guards Time (s)	Slowdown (%)	Exp. KB	Exp. KB no guards
decompress-zip	1.22	43%	240.9	63.3
downshift	1.47	3%	240.9	63.3
node-ping	4.89	22%	240.9	63.3
passport-saml	0.48	13%	0.0	0.0
requestify	3.30	12%	240.9	63.3

(a) Stubbed with static call graph

Client Proj	With guards Time (s)	Slowdown (%)	Exp. KB	Exp. KB no guards
decompress-zip	0.78	10%	16.6	3.0
downshift	1.63	13%	3.2	0.9
node-ping	4.69	19%	14.9	2.9
passport-saml	0.51	19%	0.0	0.0
requestify	3.43	15%	4.3	0.9

(b) Stubbed with dynamic call graph

Table 6.5: Results for Clients of `q` with guards enabled

have a noticeable impact on code size and execution times, and research question RQ5 aims to establish the magnitude of that effect.

We first consider performance and code size by repeating the experiments in guarded execution mode. The initial distribution sizes for the 15 applications is the same, but we noted an increase in expanded code sizes, which in many cases now exceeds the size of the original application. This is unsurprising, as the code size overhead of the guards is significant. Consider Tables 6.5a and 6.5b[10], which report experimental data for the `q` package's five clients. The first row of Table 6.5a reads: for the `decrompress-zip` client of `q`, the test suite runs in 1.22s which is a slowdown of 43% over running the test suite with the original `q` package. A performance hit is expected, as the expanded code is now running an additional conditional check around *every* function call to check if the function being called is in the specified list of dangerous functions. Moreover, during these tests 240.9KB of code is expanded, as compared to 63.25KB of code being expanded without guarded execution mode (this last column is also included in Table 6.3a). Note that, when using static call graphs, the expanded code size is almost 4x larger when guards are enabled. The performance of the code also degrades, though the raw numbers are again fairly low—we again suspect the increased slowdown to be (mostly) due to the fact that the program needs to load more code. That said, we did observe some significant overhead in longer-running applications, for instance slowdowns of 19% and 3% in the longer running test suites of `redux`'s `found` and `react-beautiful-dnd` clients, respectively (when using static call graphs), as compared with 4% and 3% respectively without guarded execution mode.

[10]The full data for all applications is included in Appendix C.

Detecting security vulnerabilities. When guarded execution mode was enabled, calls to `eval` were intercepted in running the test suites of three subject applications: `body-parser`, `send`, and `serve-static`. Upon investigation, we found that the dangerous calls were not in the code of these packages themselves, but *hidden in one their dependencies*. Specifically, all these packages rely on an *old version* of `depd` [10]: `body-parser` and `send` have a direct dependency, and `serve-static` has a transitive dependency as it depends on `send`. We confirmed that this is indeed a problem by examining the `depd` project repository on Github and found that the problematic `eval` was removed on January 12, 2018 with commit [6], which fixed three issues [7, 8, 9]. These issues were filed because `eval` is not only bad practice, but its use is disallowed in Chrome apps and Electron apps. To fix this issue, we removed the lock on the `depd` version (i.e., set it to `*`) to get the applications to use the current version of `depd`, and confirmed that all client tests still pass.

To further test the effectiveness of guarded execution mode, we ran another experiment involving two other applications with known vulnerabilities: `osenv` and `node-os-uptime`. These projects were used as experimental subjects[11] in the evaluation of a dynamic taint analysis [121] that detected vulnerabilities in them. In both projects, a function containing a call to a dangerous function (`exec` in the case of `osenv` and `execSync` in the case of `node-os-uptime`) was stubbed out by *Stubbifier*. We created a new test containing the same code fragment that was used in Karim et al. [121] to detect the vulnerability, and confirmed that the guard introduced by *Stubbifier* was triggered when the test was executed.

Guarded execution mode allows developers to detect injection vulnerabilities in imported modules of which developers may be unaware, and we found several examples of this in our experiments.

RQ6: How much does *Stubbifier* reduce the size of applications that have been bundled using `Rollup`?

To answer this research question, we conducted an experiment where we applied the the `Rollup` bundler to each subject application, and applied *Stubbifier* to the resulting bundle. Table 6.6 displays the results of this experiment. The first row of this table can be read as follows: for the `memfs` project, the size of the rollup bundle is 128KB, which is a reduction of 53% from the original size of the project. When we stubbify that bundle using the dynamic callgraph as input, the result is a bundle of 10KB, which is a further reduction

[11] Of all the subject applications considered in Karim et al. [121], these are the only two that still build, install, and have a test suite with passing tests, as required by *Stubbifier*.

	Bundle		Stubbed Bundle: Dynamic CG		Stubbed Bundle: Static CG	
Package	**Size (KB)**	**Red %**	**Size (KB)**	**Red %**	**Size (KB)**	**Red %**
memfs	128	53%	10	92%	10	92%
fs-nextra	52	0%	21	60%	21	60%
body-parser	626	36%	534	15%	534	15%
commander	72	17%	47	35%	47	35%
memory-fs	100	17%	62	38%	62	38%
glob	84	2%	42	50%	42	50%
redux	22	92%	7	67%	7	67%
css-loader	962	59%	393	59%	393	59%
q	66	77%	53	19%	53	19%
send	130	43%	89	31%	89	31%
serve-favicon	18	21%	12	31%	12	31%
morgan	54	3%	30	44%	30	45%
serve-static	107	22%	95	11%	95	11%
prop-types	**NA**	**NA**	**NA**	**NA**	**NA**	**NA**
compression	23	66%	21	7%	21	7%

Table 6.6: Effect of stubbifying bundled projects. Note: the size reduction reported in columns **Dynamic CG** and **Static CG** are *on top of* the reduction reported in the **Bundle** columns. In our experiments, *Stubbifier* achieved size reduction beyond what could be achieved with bundlers alone in all cases.

of 92% from the original bundle. When we stubbify the bundle instead with the static callgraph as input, the result is also a bundle of 10KB, with the same reduction of 92% from the original bundle.

Not all of the applications lend themselves well to bundling. For example, `commander` and `q` are configured such that when the bundler is applied, the entire package is wrapped in a single function that is called to generate the module exports. Since this function does not exist in the original module, it is not detected as reachable from the application's tests (since these exercise the original, un-bundled application). To address this, we configured *Stubbifier* to prevent it from replacing 4 functions with stubs (one in `commander`, and three in `q`) (recall from Section 6.3.2 that programmers can specify in a comment that *Stubbifier* should not stub a function or file). Beyond these, `prop-types` could not be bundled as it depends on some BabelJS libraries that throw errors when the code format is changed by the bundler, and `fs-nextra` has no dependencies so bundling it does not reduce its size at all.

That said, in every case, we see that *Stubbifier* achieves additional size reductions on applications after they are bundled, with an average of 37% further size reduction. Indeed, the purpose of bundlers is not to reduce application size, and that is merely a secondary benefit: the main goal of a bundler is to produce a single file that can be distributed for ease-of-use, and *Stubbifier* reduces the size of all of these bundles. Bundlers and *Stubbifier* are in a sense complementary.

To confirm that the debloated bundles behave as expected, we conducted an experiment in which we reconfigured the test suites of `commander`, `body-parser`, and `node-glob` to use the debloated bundle[12], and found that the project tests executed as expected with no failing tests introduced by the process.

Stubbifier achieves significant code size reductions when applied to bundled applications, by reporting a further size reduction of 37% on top of the reduction already afforded by bundlers.

6.4.3 Comparison with Mininode

Like *Stubbifier*, Mininode [126] is a tool for reducing the size of Node.js applications, but there are fundamental differences between the two tools, which we explore and evaluate in this section.

Mininode relies on a static analysis to determine code that is unused and that should be removed. Code can be removed at one of two levels of granularity: "coarse", where entire modules are removed, or "fine", where individual functions are removed. For the "fine" mode, Mininode makes use of an unsound static analysis to build a call graph of the application, using as the entry point the main file specified in the `package.json` of the project. Mininode also removes non-code artifacts such as license and configuration files.

There is a significant difference in the types of distributions used to evaluate Mininode and *Stubbifier*. In the JavaScript npm package ecosystem, a distinction is made between an application's dependencies and development dependencies: A *dependency* is another package that the application needs to function (e.g., a utility library such as lodash), whereas a *development dependency* is only needed during development (e.g., a test runner such as mocha that is needed to run the application's tests) and is not normally part of a production distribution. Mininode assumes an application's development distribution as the starting point and considers development dependencies and package tests as targets for removal. By contrast, in our work, we assume the production distribution of a package to be the starting point (which already excludes development dependencies and tests). Therefore, we do not consider development dependencies and test code when reporting results obtained with *Stubbifier*. Mininode also only supports the ECMAScript 5 version

[12] In general, adapting application test suites to work with a bundled version of the application instead of the original version can be a complex and error-prone process, as test suites may import specific functions (that may be renamed by the bundler) from specific files (that may be combined by the bundler). For the applications mentioned here, this conversion was straightforward.

of JavaScript (which dates back to 2009), whereas *Stubbifier* can debloat JavaScript ES2019 applications that use modern JS features such as modules, classes, async/await, etc.

We tried running Mininode on all 15 subject applications that we used to evaluate *Stubbifier*. Of these, 4 used features specific to JavaScript ES6+[13] causing Mininode to fail on parsing the source code; 2 of them crashed Mininode with a runtime exception[14] because Mininode dispatches a malformed call to `fs.stat`, and in 1 of them[15] Mininode removed one a production file `factory.js` file, rendering the debloated application non-functional. In the remaining 8 projects where Mininode ran successfully, we noted that the only files that it removed were development dependency module files, test files, and non-code files such as `.eslintignore` and `LICENSE`.

Fundamentally, Mininode and *Stubbifier* have different objectives and apply different techniques. Mininode completely removes code and other files such as license files. On the other hand, *Stubbifier* replaces code that is likely to be unused with stubs. Note that it is not possible to apply *Stubbifier* after applying Mininode because Mininode removes the application's tests, which *Stubbifier* needs for call graph construction. Conversely, *Stubbifier* creates production distributions that already exclude development dependences, so applying Mininode after applying *Stubbifier* does not make sense.

6.5 Threats to Validity

Our approach relies on an application's test suite as the entry point for call graph construction. This entwines the performance of our tool with the quality of the tests. An application with a low-quality test suite may generate a call graph that does not represent a comprehensive usage of the application functions, thus leading to more stubs and likely more stub expansion. To mitigate against bias, we did not consider the *quality* of an application's tests when selecting projects for our evaluation, only that the application had tests at all (and that these tests passed). Concretely, Table 6.1 shows that applications have differing numbers of tests, as many as 1706 and as few as 30, with every application having over 10K LOC. We also see a large variation in the *coverage* achieved by these test suites over the code available to be stubbed (i.e., the source code and production dependencies of an application): we see coverage as high as 99.04% and as low as 0.52%. This suggests that the quality of the test suites of the projects in our evaluation varies considerably.

[13] `memfs`,`fs-nextra`,`commander.js`,`redux`
[14] `memory-fs`,`serve-favicon`
[15] `prop-types`

Also, we are cognizant that we are drawing generalized conclusions based on a limited set of JavaScript projects. To mitigate potential bias in project selection, we selected 15 projects in a systematic manner from the most popular projects published by `npm`: from a list of projects sorted in descending order by number of weekly downloads, we attempted to install, build, and run project test suites. If a project satisfied all these criteria, we then randomly selected from its clients and attempted to install, build, and run their tests; if the project had five such clients, it was selected. We also note that the subject applications vary considerably in size, in number of dependencies, as well as application domains: e.g., `memfs` is an in-memory file system, `body-parser` is a parser for request bodies, and `css-loader` is a custom loader for `css` files. In a similar vein, we are cognizant of the fact that the five chosen client applications might not be representative clients of the projects. To mitigate potential bias here, we chose the clients randomly, and we chose five of them to try and get a variety of use cases. We note that there is a range in the amount of code loaded dynamically across the clients, so we see that not all the clients use the same features of a package. We do also note that there is often overlap in the stubs expanded across clients: this is unsurprising, as we expect some overlap in the ways clients use a project, and it indicates untested functionality in the project.

It is also possible that the reported runtimes are subject to measurement bias. We mitigate this by running all performance experiments on a machine with no other processes running. We also report the average run time over 10 runs, after discarding two initial runs, which minimizes risk of long experiment startup time.

In our experiments with the Rollup bundler, we had to manually configure *Stubbifier* to avoid stubbing four functions in the bundles for `commander` and `q` that were introduced by the bundler. Since these functions did not occur in the call graphs created by *Stubbifier*, they would otherwise have been replaced with stubs, resulting in size reductions in excess of 95%. However, such a size reduction would have been counterproductive—these functions are always executed when the bundles are used, and thus the introduced stubs would always have to be expanded. There is a potential for human error here, but identifying these four functions was not difficult: for `commander`, the bundler wrapped the entire module in an immediately invoked function expression (IIFE), and in the case of `q` the bundler included large swaths of code in the exported object of the bundle. Longer term, an automated solution to this problem could be devised.

6.6 Relation to Previous Work

Our work was inspired by DOLOTO [138], a tool that applies code-splitting to an application based on "access profiles" obtained from users interacting with an instrumented version of the application. These access profiles define clusters of functions that should be loaded together, and functionality that should be part of the distribution of an application. Applications processed by DOLOTO ship with enough functionality for initialization, and inessential functions are replaced with small stubs that are either replaced once their original code is loaded lazily, or on-demand when a stubbed function is invoked.

There are several factors that make *Stubbifier* more practical than Doloto. Most importantly, *Stubbifier* is fully automatic, debloating an application based on call graphs that were constructed from its tests. *Stubbifier* handles JavaScript ECMAScript 2019 [74], which includes many features (e.g., classes, promises, async/await, generators, modules etc.) that were not present when Doloto was developed in 2008. Moreover, *Stubbifier* supports not only the function-level stubs that were used by Doloto, but also file-level stubs to handle the common case where all functions in a file are found to be unreachable. *Stubbifier* also provides a guarded execution mode, which prevents injection vulnerabilities resulting from calls to functions such as `eval` and `exec` when they are invoked from within untested code that resulted from expanding stubs. Lastly, *Stubbifier* has been developed to be used in conjunction with bundlers.

In Section 6.4.3, we compared *Stubbifier* with Mininode [126], another tool for debloating Node.js applications and noted significant differences between the two debloating techniques: (1) Mininode targets development distributions, which include application tests as well as dependencies only needed during development (e.g., test suite runners like `jest`), whereas *Stubbifier* targets production distributions, which already exclude tests and development dependencies; (2) Mininode completely removes code, and can introduce application crashes if the removed code is called, whereas *Stubbifier* replaces code with stubs that can fetch the original code as needed; (3) Mininode targets the ECMAScript 5 version of JavaScript, which lacks many widely used features such as classes, async/await, and modules, and these are all supported by *Stubbifier* which supports the ECMAScript 2019 version of JavaScript. We ran Mininode on the 15 subject applications in the evaluation, and found that Mininode successfully debloated only 8 of them, and in those it only removed development dependencies, test files, and non-code files. There is also recent work on debloating other languages: JShrink [56] is a tool for debloating the bytecode of Java applications. Their technique makes use of a combination of both static and dynamic analyses, to use both the strong type guarantees of the Java language, and to also deal with dynamic language features that are becoming more prevalent in modern Java use.

Section 3.4 broadly discussed debloating, which is pertinent in this chapter. Building minimal application bundles is both well-studied and prevalent in industry. Many of these approaches discussed rely on some form of "application profile" obtained via program analysis—*Stubbifier* builds this profile via static or dynamic analysis of application tests. Trimming optional functionality from applications has been studied by many, as well as entirely removing unused code., Finally, *Stubbifier* relies on code splitting, though the primary purpose of code splitting is to remove optional functionality until it is needed, whereas it is leveraged in this approach for aggressive dead code elimination.

6.6.1 Control Flow Integrity

The guarded execution mode resembles works on Control Flow Integrity (CFI) verification by, e.g., Abadi et al. [32]. A CFI policy dictates that program execution must follow a predetermined path of a control flow graph, enforced via program rewriting and runtime monitoring. Conceptually, our guarded execution mode enforces a policy where program execution cannot invoke a predefined list of functions. Zhang et al. [252] present a CFI approach that enforces a policy preventing jumps to any but a white-list of locations, whereas our guarded mode enforces a black-list of functions. Niu and Tan [169] develop a "per-input" CFI technique to avoid the overhead of constructing a control flow graph, and our mode avoids this altogether by pre-transforming code to intercept calls.

6.6.2 Vulnerability Detection and Reduction

Guarded execution mode's ability to intercept dangerous function execution in dynamically loaded code is intended to reduce the attack surface of applications, and ultimately make them less vulnerable to attacks.

There is a wealth of existing work in this area. On the topic of traditional injection vulnerabilities, Gauthier et al. [95] describe an approach for detecting injection vulnerabilities through a mix of white-box analysis (of application code), and black-box analysis (of third party modules), and Nielsen et al. [167] present a static dataflow analysis tool which overcomes scalability issues by analyzing a limited amount of third-party modules. Taint analysis is a popular method for detecting these types of vulnerabilities, and Staicu et al. [202] describe an approach to automatically extracting taint specifications for JavaScript libraries, which is important as taint analysis require taint specifications to report taint flows, and manually coming up with taint specifications is tedious at best, and error-prone at worst. Injection vulnerabilities are not alone in plaguing JavaScript code, and Li et

al. [131] present a novel data structure constructed from various static analysis, model a variety of vulnerabilities (e.g., beyond injection), and use abstract interpretation to detect them. Node.js allows JavaScript programs to execute arbitrary shell commands, and Vasilakis et al. [222] detail an approach specifying read-write-execute permissions for third-party libraries, noting that much third-party code executes with more elevated permission than is required. Further, Staicu et al. [201] report on a study of over 200k Node.js applications, arguing that command-line injection vulnerabilities are common, and present a system that synthesizes grammar-based policies from template values (that are generated as abstractions of values likely to result in vulnerabilities). Another interesting vulnerability specific to languages with prototype-based inheritance (like JavaScript) is reported on by Li et al. [130]; known as *prototype pollution*, base object prototypes are modified to introduce new attack vectors.

The `npm` ecosystem does provide a "security audit" of packages it installs, and typically reports that dozens of vulnerabilities exist in installed dependencies. Zimmermann et al. [253] conduct a study of security threats in the `npm` ecosystem, and determine that a lack of maintenance contribute to many present vulnerabilities. Updating packages is important to keep up with security patches, and semantic versioning helps developers determine the work involved in downloading a new version of a package; Møller and Torp [160] argue that semantic versioning is poorly used in JavaScript, and propose a technique to detect breaking changes in security patches using fuzz testing of API models.

6.7 Conclusion

JavaScript is an increasingly popular language for server-side development, thanks in part to the Node.js runtime environment and the vast ecosystem of modules available on `npm`. Unfortunately, `npm` installs modules with *all* of their functionality, even if only a fraction is needed, which causes an undue increase in code size. In this chapter, we presented a fully automatic technique that identifies dead code by constructing static or dynamic call graphs from the application's tests, and replaces code deemed unreachable with either file- or function-level stubs that can fetch and execute the original code dynamically. The technique also gives users the option to guard their applications against injection vulnerabilities in untested code that result from stub expansion. This technique is implemented in a tool called *Stubbifier*, which supports the ECMAScript 2019 standard.

In an empirical evaluation on 15 Node.js applications and 75 clients of these applications, *Stubbifier* reduced application size by 56% on average while incurring only minor performance overhead. The evaluation also showed that *Stubbifier*'s guarded execution

mode is capable of preventing several known injection vulnerabilities that are manifested in stubbed-out code. Finally, *Stubbifier* works alongside bundlers, and for the subject applications under consideration, we measured an average size reduction of 37% in distributions produced by bundlers.

Future work includes the application of similar debloating techniques to other programming languages. A key enabling factor for our technique is the availability of a mechanism for executing arbitrary code at run time, similar to JavaScript's `eval` feature. While such mechanisms tend to create significant challenges for sound static analysis, they enable the implementation of stubs that load missing code at run time. The use of a fast program analysis techniques that generate an unsound call graph is generally also well suited for dynamic languages.

6.8 Discussion

In this work, we propose to use *code splitting* with unsound analysis as an alternative to dead code elimination using sound analysis. Instead of outright removing dead code (requiring precise analysis to be effective), we apply code splitting to code that unsound analysis finds to be likely dead. A key observation underlying this work is that far more JavaScript code is dead than it appears.

It is worth investigating scenarios where it is not catastrophic to wrongfully apply an optimization. Dynamic languages in particular benefit from such approaches as it is rare to be sure about anything when analyzing them. Just-in-time (JIT) compilers already do this to some degree through *speculative optimization*, where a JIT will speculate about properties of code blocks and optimize them accordingly, and if ever the assumptions are invalidated the code is "de-optimized" and the original code is executed. In this work, we speculated on the liveness of code, and if ever we were wrong about the code being dead it was fetched and executed as if it were never removed. It is worth investigating optimizations predicated on speculation of richer properties of code or of values: e.g., dispatching based on the sortedness of a list, or on the upper-triangularity of a matrix.

As a concrete avenue for future work, the dynamic loading mechanism presented in this chapter could be improved by devising an analysis to automatically build "usage profiles" for the application being debloated. For example, consider the snippet in Figure 6.7. Here, if both `foo` and `bar` were determined to be likely dead, then they would both be replaced with stubs. Then, if `foo` were called, the code for `foo` would be fetched and executed, which includes a call to `bar`, which would trigger a separate dynamic load.

```
359  function foo() {
360    // big code
361    bar()
362  }
363
364  function bar() {
365   // big code
366  }
```

Figure 6.7: Example of inefficient dynamic loading in applications debloated with *Stubbifier*.

It would instead be more efficient to load the entire "usage profile" of `foo`, which includes the code for `bar`, when `foo` is loaded dynamically. That being said, the overhead associated with dynamic loading is relatively small, although improving that is sure to make this approach more appealing in general. Further, unused functionality could be communicated directly to developers so they can be informed about what their test suites are missing (e.g., the common sets of unused functionality in `redux`).

Chapter 7

Lazy Loading

Abstract

Front-end developers want their applications to contain no more code than is needed in order to minimize the amount of time that elapses between visiting a web page and the page becoming responsive. However, front-end code is typically written in JavaScript, the ubiquitous "language of the web", and tends to rely heavily on third-party packages. While the reuse of packages improves developer productivity, it is notorious for resulting in very large "bloated" applications, resulting in a degraded end-user experience. One way to combat such bloat is to *lazily load* external packages on an as-needed basis, for which support was added to JavaScript in 2020 when *asynchronous, dynamic* imports were added to the language standard. Unfortunately, migrating existing projects to take advantage of this feature is nontrivial, as the code changes required to introduce asynchrony may involve complex, non-local transformations.

In this work, we propose an approach for automatically introducing lazy loading of third-party packages in JavaScript applications. Our approach relies on static analysis to identify external packages that can be loaded lazily and generates the code transformations required to lazily load those packages. Since the static analysis is unsound, these transformations are presented as suggestions that programmers should review and test carefully. We implement this approach in a tool called *Lazifier*, and evaluate *Lazifier* on 10 open-source front-end JavaScript applications, showing that each application was successfully refactored, reducing initial application size and load times in all cases. On average, for these applications, *Lazifier* reduces initial application size by 36.2%, initial load time by 29.7%, and unsoundness did not arise in any of these applications.

7.1 Introduction

In web application development, it is highly desirable to minimize the time it takes for an application to load and become responsive [94, 135, 137, 57]. Therefore, developers generally aim to keep the size of their distribution as small as possible and rely on tools such as bundlers, minifiers, and tree-shakers [64, 157, 186, 232] to minimize code size. Unfortunately, such tools are of limited use in scenarios where an application contains functionality that is (potentially) required, but not immediately on application startup. In such cases, responsiveness can be improved by loading the code associated with such functionality asynchronously, if or when its first use occurs.

In this work, we propose an approach for automatically refactoring applications to introduce lazy loading. We are targeting a specific scenario where the functionality to be loaded lazily is isolated in a third-party library that is imported by the application under consideration. Our approach relies on static analysis to identify packages that are only used in the context of event-handling code, as they are likely only needed conditionally (or at least not needed on startup). Then, for each of these packages, another static analysis establishes the extent of the code that needs to be modified to accommodate asynchronous, lazy loading of the package. Finally, a set of declarative rewrite rules specifies the code changes required to transform the application.

We implemented this approach in a tool called *Lazifier* that targets the JavaScript programming language (ECMAScript 2021). Similar to recent other refactoring tools [49, 216, 100], *Lazifier* employs unsound static analysis, so the proposed code transformations are presented as *suggestions* that programmers should review and test carefully before applying. In an experimental evaluation on 10 open-source client-side JavaScript applications, the code transformations proposed by *Lazifier* resulted in an average initial application size reduction of 36.2%, which caused applications to speed up initial load time by 29.7% on average. Furthermore, we found that the actual lazy loading of packages affected by the transformations incurs little overhead. Finally, despite the potential for unsoundness in the static analysis, we found that none of the transformations proposed by *Lazifier* for the 10 subject applications caused unwanted behavioral differences.

In sum, this chapter contains:

- an automated approach for identifying packages that can be loaded lazily, and a set of rewrite rules specifying how to refactor an application to load those packages lazily;
- an implementation of this approach in a tool called *Lazifier*, targeting the JavaScript programming language;

- an evaluation of *Lazifier* on 10 applications that suggests that *Lazifier* reduces initial application size (36.2%, on average) and load time significantly (29.7%, on average) with little overhead associated with dynamic loading.

The remainder of this chapter is organized as follows. First, the relevant background is covered in § 7.2, the problem is further motivated in § 7.3, the approach is described in-depth in § 7.4 (in which the implementation of our tool, *Lazifier*, is overviewed in subsections 7.4.4), followed by the evaluation in § 7.5, threats to validity in § 7.6, the work is positioned with respect to related literature in § 7.7, and § 7.8 concludes. A discussion follows in § 7.9 relating the work in this chapter with the rest of the thesis.

7.2 Background

Refer to Chapters 2.2.1 and 2.2 for background on asynchrony in JavaScript, and Chapter 2.3.1 for more information regarding how to import external functionality into a JavaScript application.

7.3 Lazy Loading

To illustrate our approach, consider an open-source JavaScript application that displays a list of recent movies to users, complete with information about them (**Movies-web-ui** [33]). Users can filter the list of movies and, optionally, export their filtered selection. The code snippet in Fig 7.1**(a)** is taken directly from **Movies-web-ui**, showing how they implement an "export" button and associated functionality. Note that this application uses a few external packages: `React`, an extremely popular UI framework for JavaScript, `file-saver` [77] for saving files, and `xlsx` [196] for dealing with spreadsheet-like data. The file exports a function `exportCSV` that creates a JSX [1] button component (lines 382-386). The `"click"` event handler associated with this button (lines 383-384) eventually calls the `exportToCSV` function (lines 374-380), which leverages the `xlsx` package to convert a JSON file representing the user's selection to a sheet (line 375), and `file-saver` to save the selection to a file (line 379).

Crucially, in this example, the `xlsx` and `file-saver` packages are only needed to implement the export functionality and are not useful to users that simply want to browse

[1] JSX is a type provided by React that closely matches HTML, allowing programmers to easily construct HTML-like objects in their JavaScript code.

```
import React from 'react';
import * as fileSaver from 'file-saver';
import * as xlsx from 'xlsx';

export const exportCSV = ({csvData, fileName}) => {
  const fileType = '...';
  const fileExtension = '.xlsx';
  const exportToCSV = (csvData, fileName) => {
    const ws = xlsx.utils.json_to_sheet(csvData);
    const wb = {Sheets: {...}, SheetNames: [...]};
    const buffer = xlsx.write(wb, {...});
    const data = new Blob([buffer], {type: fileType});
    fileSaver.saveAs(data, fileName + fileExtension);
  }
  return (
    <button className="export"
      onClick={(e) =>
        exportToCSV(csvData,fileName)}>
      Export
    </button>
  )
}
```

(a)

```
import React from 'react';
// this import was removed
// this import was removed

export const exportCSV = ({csvData, fileName}) => {
  const fileType = '...';
  const fileExtension = '.xlsx';
  const exportToCSV = async (csvData, fileName) => {
    const fileSaver = await import('file-saver');
    const xlsx = await import('xlsx');
    const ws = xlsx.utils.json_to_sheet(csvData);
    const wb = {Sheets: {...}, SheetNames: [...]};
    const buffer = xlsx.write(wb, {...});
    const data = new Blob([buffer], {type: fileType});
    fileSaver.saveAs(data, fileName + fileExtension);
  }
  return (
    <button className="export"
      onClick={async (e) =>
        await exportToCSV(csvData,fileName)}>
      Export
    </button>
  )
}
```

(b)

Figure 7.1: Excerpt of a client-side application which uses xlsx: (a) version with static import (b) version with dynamic import

the list of movies. It should also be noted that the references to these packages on lines 375, 377, and 379 are the only references to these packages in the entire application.

In such cases, it is desirable to load packages lazily, so that users who do not use the associated functionality do not incur the overhead of loading code that they will not use. The code snippet in Fig 7.1**(b)** depicts how this can be achieved, and code changes are highlighted. First, note the lack of static imports to `xlsx` and `file-saver`, and the inclusion of dynamic imports to the packages instead (lines 397-398).

The call `import('file-saver')` on line 397 creates a promise that is resolved with an object representing the `file-saver` package. Once the loading of the package has been completed, the `await` on the same line ensures that this object can be assigned to the local variable `fileSaver`. Recall that `await` expressions are only allowed in the context of `async` functions, so the `exportToCSV` function must gain the `async` keyword (line 396). This changes the return type of `exportToCSV` to $Promise\langle JSX\rangle$, so all call sites to this function should be `await`-ed to ensure that application behavior remains unchanged. In particular, an `await` is added at the call to `exportToCSV` on line 408. This new `await` requires the surrounding function to be made `async` as well (line 407), at which point we have reached a context that implicitly handles asynchrony: callbacks that serve as event handlers are not expected to return anything, so no further transformations are required once they are made `async`.

This simple refactoring reduces the amount of code that is loaded by over 30% (from 1.4mb to 0.96mb), and improves the initial load time of the application by just under 50% (from 517ms to 286ms, averaged over 10 runs). If the user *does* want to export their selection, the packages are loaded rather quickly (0.11s), and the total amount of code loaded by the application is 1.4mb, i.e., the same as the original size.

There are certain additional complexities that the above example only hinted at. For instance, when making a function `async`, *all* call sites to the function must be `await`-ed, no matter where they are. This can cause a cascade of transformations that may not be localized to a single file. Further, certain code patterns need to be modified to accommodate `async` functions (e.g., the expression `someArray.forEach(f)` is blocking if the callback `f` is synchronous, but non-blocking if `f` is `async`). In the next section, we describe these complexities and present our approach to automatically detecting packages that can be loaded lazily, and specify the code transformations required.

7.4 Approach

Our approach for automatically refactoring applications to introduce lazy loading consists of the following three steps:

1. *Determine packages that are only used in the context of event handlers;*
2. *Confirm which of these can be loaded lazily, and identify the required transformations;*
3. *Enact the transformation.*

For (1), a static analysis detects which packages are *only* used in the context of event handling code and not initially needed by the application. For (2), another static analysis determines all of the functions containing references to a given lazy loading candidate. Each of those functions will require a dynamic, *asynchronous* import of the package, which will require several other code transformations to support the now asynchronous import. If any of these transformations are not possible, the lazy loading candidate is discarded. Finally, for (3) a set of declarative rewrite rules describes the code changes required to refactor the application to lazily load the package.

Soundness. We assume that the static analyses used in steps 1) and 2) are potentially *unsound*, because sound, precise, and scalable static analysis for JavaScript is well beyond the state-of-the-art [175, 125, 130]. Thus, the transformations proposed by the approach may not preserve behavior, and should be carefully reviewed by a programmer, similar to the approach taken by other refactoring tools for JavaScript [49, 216, 100]. In Section 7.5, we investigate the degree to which this unsoundness causes behavioral differences.

7.4.1 Identify Candidate Packages for Lazy Loading

To identify packages that should be loaded lazily, we provide a fully-automated analysis that detects packages that are only used in the context of event-handling code. Given a call graph for an application, this analysis identifies functions that are supplied to event-handling mechanisms (e.g., registered as "on-click" attributes of HTML elements, or registered as event listeners), and determines all of the functions that are (transitively) called from those handlers. If *all* references to a package are in this list of functions, then it is flagged as being a candidate for lazy loading. This list of event handlers is:

- functions passed to `onClick` or other `on` or `click` events on JSX and HTML components, including functions identified using string representations of their name;
- any code snippets included in an event handler attribute (e.g., code in the `onClick` event of an HTML element);
- functions passed as callback arguments to event handlers (e.g., `reader.on('load', callback)`);
- functions assigned to properties of the `window` object that represents the Document Object Model (DOM).

7.4.2 Validate and Determine Transformations Required

To successfully load a package p lazily, all static imports to p must be removed, and functions containing references to p must be refactored to load the package dynamically. This involves removing static `import ... from 'p'` statements and inserting dynamic `import('p')` expressions where appropriate. The expression `import('p')` yields a promise that eventually resolves with the content of the package `'p'`. While that promise is pending, the current context that depends on the package should not proceed, and `await`-ing that call will suspend execution until the promise is resolved. Then, if assigning the `await`-ed import to a variable (e.g., `let x = await import('p')`), the package itself will be stored in `x` and execution can resume.

Now, `await` expressions are only allowed inside of functions marked as `async`, but making a function `async` changes its return type to $Promise\langle T\rangle$, where T is the function's original return type. To preserve existing application behavior, all call sites to this function will need to be `await`-ed, which itself requires more functions to be made `async` and more call sites to be `await`-ed, and so on. It is imperative that *all* call sites to newly `async` functions be `await`-ed, else program behavior will be affected; this means that the transformation is *all or nothing proposition*, and if any call sites cannot be `await`-ed, we must abandon the entire transformation, and discard p as a lazy loading candidate.

Algorithm 3 describes the process of creating the set S_{async} of functions needing to be made `async` while validating the transformation. As inputs to the algorithm, the package p is supplied along with the call graph CG of the program. First, S_{async} is initialized as the empty set (line 1), and the list F of functions yet to be processed is initialized with all functions containing references to the package p (line 2). The main loop (lines 3-15) iterates through functions $f \in F$ that have not yet been visited. First, lines 6-8 describes a

Algorithm 3: Validating p and building S_{async}

```
   Data: p: a package being imported dynamically
   Data: CG: the call graph of the program
 1 let S_async := {};
 2 let F := [functions referencing p];
 3 while F not empty do
 4     let f := select and remove a function from F;
 5     if f not visited then
 6         if f is a reaction or f is argument to promise constructor or f registered
             as event handler then
 7             S_async := S_async ∪ {f};
 8             continue;
 9         let C_f := callers of f in CG;
10         if f is constructor or c ∈ C_f is top level or f returns promise then
11             S_async := {};
12             break;
13         S_async := S_async ∪ {f};
14         F := F ∪ C_f;
15         mark f as visited;
16 return S_async;
```

special case where a function to be made asynchronous is already in a context that handles asynchrony, in which case no further transformations are required. Then, all callers of the function f are obtained from the call graph (line 9). Lines 10-12 *validates* the transformation by identifying situations that cannot support asynchrony. First, constructors cannot be `async`. Second, if f is called at the top level of the application, there is no sense in lazily loading p as the dynamic import would be executed on application startup anyway. (Also, top-level `await` expressions are only supported as of ECMAScript 2022.) Third, if f already returns a promise, the programmer is likely using it accordingly and may not want calls to it to be `await`-ed, and so it should not be transformed. In such cases, the transformation is rejected and p is not loaded lazily. If f passes this check, then f is added to S_{async}, all of f's callers are added to the list F of functions left to process, and f is marked as visited; analysis continues until F is exhausted.

7.4.3 Code Transformations

The application can be refactored to lazily load package p once the set S_{async} of functions that need to be made `async` is known. Several transformations are required to handle the transition to asynchronous imports, specified as declarative rewrite rules in Figure 7.2. The figure depicts simplified, idealized JavaScript to illustrate the salient details of the transformation. We will describe them one by one next.

Async-Function: This transformation is simple: if a function `f` is in the set S_{async} of functions that need to be made `async`, the function definition gains the `async` keyword.

Async-Call: All potential calls to a function `f` $\in S_{async}$ need to have `await` expressions inserted before the call.

ForEach-ForOf: The expression `arr.forEach(f)` calls the callback `f` on each element of `arr`, and importantly *returns nothing*, i.e., `forEach` is type *void*. If `f` were made asynchronous, the call to `forEach` would not wait for all of the asynchronous calls to resolve, and execution would simply continue past the call. In the event that `f` contains no `return` statements, the body `B` of `f` is made into the body of a `for ... of` loop that iterates over the elements of the array (the loop iterator `a` is chosen to match the argument name of `f`).

ForEach-Map: In the event that `f` *does* contain a `return` statement, conversion to a `for ... of` loop is not possible. Instead, the `forEach` is transformed into a `map`, and the

$$\frac{\texttt{f} \in S_{async}}{\texttt{fun f(A) \{B\}} \longrightarrow \texttt{async fun f(A) \{B\}}} \quad \text{(Async-Function)}$$

$$\frac{\texttt{f} \in S_{async} \qquad \texttt{g} \text{ can resolve to } \texttt{f}}{\texttt{g(args)} \longrightarrow \texttt{await g(args)}} \quad \text{(Async-Call)}$$

$$\frac{\begin{array}{c}\texttt{f} \in S_{async} \qquad B \text{ body of } \texttt{f} \\ \text{no returns in } B \qquad a = \text{the single argument of } \texttt{f}\end{array}}{\texttt{arr.forEach(f)} \longrightarrow \texttt{for}([\texttt{i}, a] \texttt{ of arr.entries}())\ \{B\}} \quad \text{(ForEach-ForOf)}$$

$$\frac{\begin{array}{c}\texttt{f} \in S_{async} \qquad B \text{ body of } \texttt{f} \\ \text{returns in } B\end{array}}{\texttt{arr.forEach(f)} \longrightarrow \texttt{await Promise.all(arr.map(f))}} \quad \text{(ForEach-Map)}$$

$$\frac{\texttt{f} \in S_{async}}{\texttt{arr.map(f)} \longrightarrow \texttt{await Promise.all(arr.map(f))}} \quad \text{(Await-Map)}$$

$$\frac{\begin{array}{c}p \in P_D \qquad v_0, ..., v_n \textbf{ ref } p \in \texttt{B} \\ \texttt{dynImp} := \texttt{const } p_{name} = \texttt{await import}(p) \\ \texttt{decl}_k := \texttt{const } v_k = p.v_k^{name} \quad \forall k \in 0, ..., n\end{array}}{\texttt{fun f(A) \{B\}} \longrightarrow \texttt{fun f(A) \{dynImp; decl}_0\texttt{; ... decl}_n\texttt{; B\}}} \quad \text{(Insert-Dynamic-Import)}$$

$$\frac{\texttt{x} \in S_{async} \qquad \texttt{f}_\texttt{B} := \texttt{async } () \Rightarrow \{\texttt{B}\}}{\texttt{get x() \{B\}} \longrightarrow \texttt{get x() \{return f}_\texttt{B}\texttt{();\}}} \quad \text{(Getter)}$$

Figure 7.2: Transformation rules for introducing lazy loading and necessary code changes to support newly introduced asynchrony.

call to `map` is surrounded in an `await`-ed `Promise.all` to ensure that all of the asynchronous callbacks fully execute before continuing.

AWAIT-MAP: Similar to the previous rule, if a callback passed to `map` is to be made asynchronous, the `map` is surrounded in an `await`-ed `Promise.all`.

INSERT-DYNAMIC-IMPORT: If a function `f` contains references $(v_0, ..., v_n)$ to a package p that is to be made dynamic ($p \in P_D$), a dynamic import to the package p is created ($\texttt{const}\ p_{name} = \texttt{await import}(p)$), where p_{name} will serve as a reference to the package in this scope. Then, declarations are created for each $v_k \in v_0, ..., v_n$ extracting the relevant component v_k^{name} from the import p_{name}. The dynamic import and associated declarations are then inserted at the beginning of the function body.

GETTER: Getters present a special case as they cannot be made asynchronous. A new asynchronous function $\texttt{f}_\texttt{B}$ is created with the body `B` of the getter `x`. The body of `x` is then replaced with a return to the call to $\texttt{f}_\texttt{B}$—callers of `x` will `await` calls to it, and so the promise returned by $\texttt{f}_\texttt{B}$ can be `await`-ed then.

The code transformation in the motivating example was determined automatically using this approach, and involved applications of rules ASYNC-FUNCTION, ASYNC-CALL, and INSERT-DYNAMIC-IMPORT. Fig. 7.3 shows small code examples depicting the transformations associated with the other rules: Fig. 7.3**(a)** and **(b)** shows rule FOREACH-FOROF, Fig. 7.3**(c)** and **(d)** shows rule FOREACH-MAP, Fig. 7.3**(e)** and **(f)** shows rule AWAIT-MAP, and finally Fig. 7.3**(g)** and **(h)** shows rule GETTER.

7.4.4 Implementation

This approach is implemented in a tool called *Lazifier*. All static analyses are built in CodeQL [152], including data flow analyses required to detect uses of imported packages and call graph construction. All call graphs were obtained through CodeQL's own static call graph construction algorithm for JavaScript [154], which is unsound. The code transformation is built in JavaScript using Babel [52] to parse code, manipulate ASTs, and emit transformed code.

```
arr.forEach((e) => {
  if (e)
    foo();
  else
    bar();
});
```

(a)

```
for([i, e] of arr.entries()) {
  if (e)
    await foo();
  else
    bar();
}
```

(b)

```
arr.forEach((e) => {
  if (e)
    return foo();
  else
    return bar();
});
```

(c)

```
await Promise.all(arr.map(async (e) => {
  if (e)
    return await foo();
  else
    return await bar();
}));
```

(d)

```
arr.map((e) => {
  if (e)
    foo();
  else
    bar();
});
```

(e)

```
await Promise.all(arr.map(async (e) => {
  if (e)
    await foo();
  else
    await bar();
}));
```

(f)

```
const o = {
  x : 1,
  get y() {
    return foo(x);
  }
}

o.y;
```

(g)

```
const o = {
  x : 1,
  get y() {
    return (async () => {
      return await foo(x);
    })();
  }
}

await o.y;
```

(h)

Figure 7.3: Code showing the before and after of applying select rewrite rules: **(a)-(b)** shows ForEach-ForOf, **(c)-(d)** shows ForEach-Map, **(e)-(f)** shows Await-Map, and **(g)-(h)** shows Getter.

Table 7.1: Information about subject applications. The first row reads: *the first application is called* ***upoint-query-builder****, and commit hash* `f9aa0f1` *was used for the evaluation;* ***upoint-query-builder*** *has 10,341 lines of code. The initial size of the application is 0.84mb, reduced to 0.61mb after loading modules lazily, corresponding to a 27.4% size reduction. The size of the application once modules are loaded dynamically is 0.84mb. It took 201s to run Lazifier on this project, which required an additional 28s to build the CodeQL database.*

Project Name	Commit Hash	LOC	Sizes (mb)				Run Time (s)	
			Before	After	% Red.	Exp.	Tool	QLDB
upoint-query-builder [107]	`f9aa0f1`	10,341	0.84	0.61	27.4%	0.84	201	28
excelreader [190]	`4a5f9cb`	9,733	4.8	3.4	29.2%	4.8	187	44
task [87]	`b641bc0`	9,747	0.94	0.48	48.9%	0.94	180	36
react-excel [109]	`2d59e85`	9,685	1.9	1.5	21.1%	1.9	178	33
Movies-web-ui [33]	`58904a3`	9,789	1.4	0.96	31.4%	1.4	180	35
ExcelSheet_Validation_Reactjs [223]	`f38cb9e`	9,942	0.90	0.40	55.6%	0.90	181	35
scrambles-matcher [208]	`1de93f7`	11,304	1.1	0.83	24.5%	1.1	188	37
timetable [111]	`0fa8527`	9,932	0.60	0.38	36.7%	0.60	314	80
workday-schedule-exporter [38]	`97ca596`	9,718	0.90	0.44	51.1%	0.90	186	35
react-excel-csv [221]	`18c6d97`	9,779	0.85	0.62	27.1%	0.85	206	34
		Avg. Size Reduction:			36.2%	**Avg. Time:**		240

7.5 Evaluation

We pose the following research questions in order to evaluate the approach proposed in this chapter:

RQ1. How does lazy loading affect the size and initial load time of applications?

RQ2. How often does the transformation introduce unwanted behavioral changes?

RQ3. How much code is loaded lazily, and how quickly is it loaded?

RQ4. How many code changes are required to support lazy loading?

RQ5. What is the running time of *Lazifier*?

Experimental Methodology

To answer these research questions, we first compiled a list of 10,000 open-source client-side JavaScript applications by scraping GitHub for repositories that had JavaScript UI frameworks stated as dependencies. Then, we ran the `npm-filter` [50] tool to identify projects for which *Lazifier* identified at least one package as a candidate for lazy loading (yielding 998 projects). We manually inspected projects in this list until we found 10 that could

be successfully installed, started, and interacted with. The vast majority of JavaScript projects on GitHub suffer from installation errors (e.g., developer-specified dependencies no longer work), build errors (e.g., build configurations that are only valid for certain operating systems/environments), or environment errors (e.g., many client-side applications rely on external servers that are inaccessible). Since we wanted to have a high degree of confidence in our understanding of our subject applications, we expended considerable effort finding applications that suffered from none of these aforementioned issues.

To answer **RQ1**, we first determine the original application's initial size using the "bytes transferred" metric from Chrome DevTools' [102] "Network" tab on a hard refresh of the application page, and then apply the transformation and similarly determine the initial size of the transformed application. To time the initial application load, we again leverage the Chrome DevTools' "Network" tab, and note the "Load" time field upon performing a hard refresh—we note this time pre- and post-transformation, and collect and average 10 load times.

To answer **RQ2**, we manually interacted with each application to determine how to make it execute code from packages that were flagged to be loaded lazily, then applied the transformation and repeated the interaction, manually ensuring that the application behavior was unchanged.

To answer **RQ3**, we identify how to trigger each of the dynamic imports (in the same manner as in **RQ2**), and note the size of the code chunk transferred when doing so through the Chrome DevTools' "Network" tab (again consulting the "bytes transferred" metric), and note the time taken to transfer that chunk through the "Load" time field.

To answer **RQ4**, we configured *Lazifier* to: display which packages were flagged to be loaded lazily, display the dynamic import statements that were added to the program, and log the code transformations it was applying.

And finally, to answer **RQ5**, we used the Unix `time` utility to time the execution of *Lazifier* on each application. To run *Lazifier*'s analyses, a CodeQL database must be built for the project, and so we used the time utility to time the CodeQL database build for each project.

All measurements were taken on a 2016 MacBook Pro running Catalina 10.15.7, with a 2.6GHz Quad-Code Intel Code i7 processor and 16GB RAM. We used Google Chrome version 112.0.5615.137 (Official Build) (x86_64) in incognito mode. Next, we respond to each of the **RQ**s in turn.

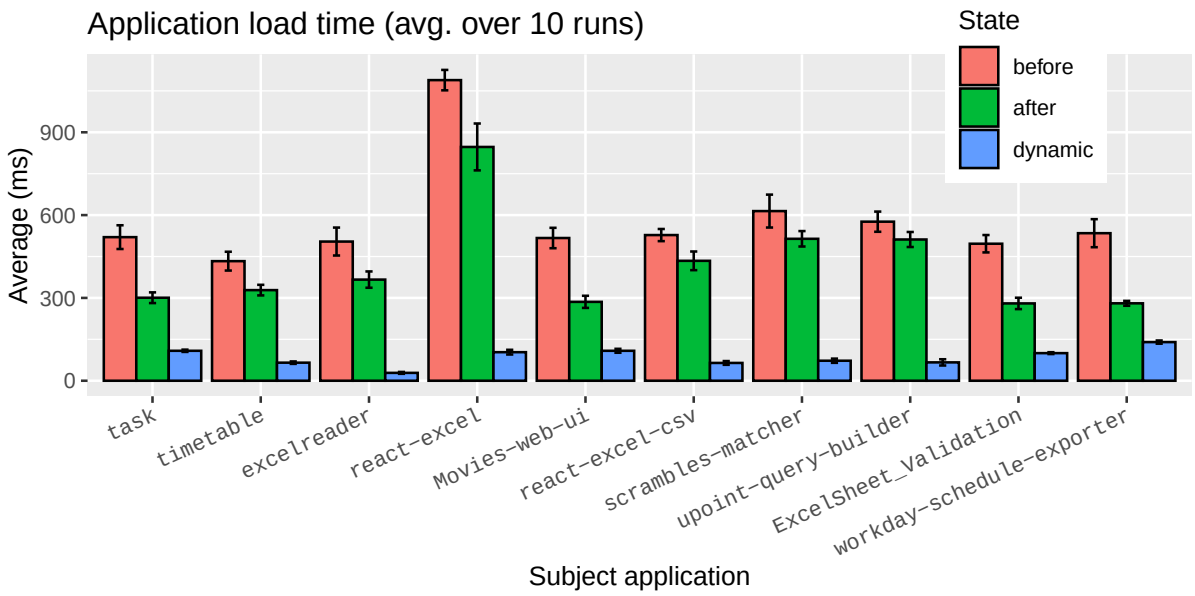


Figure 7.4: Load times for each subject application are depicted in this plot, with a set of three columns for each application. In each set, three times are presented: first, the time taken pre-refactoring (before), then after refactoring (after), and finally the time taken to dynamically load all packages (dynamic). These are averages over 10 runs, and error bars indicate +/- one standard deviation.

RQ1: How does lazy loading affect the size and initial load time of applications?

Lazifier's transformation leverages ECMAScript 2020's ability to load packages on demand: If all static imports to a package are replaced with dynamic imports, the JavaScript runtime dynamically fetches the package when a dynamic import is executed, and the package is not included in the application at start time. The initial application size is reported in columns **Initial Size (mb) Before** and **After** in Table 7.1, corresponding to the size of the applications pre- and post-refactoring. We note significant size reduction across all applications (36.2% on average), as high as 51.6%.

While smaller applications are desirable in and of themselves, the speed at which an application starts is also important to users. We investigate the degree to which this size reduction hastens the initial load time of refactored applications. Averages of 10 load times are reported in Fig. 7.4, with three columns for each subject application, the first two of which are relevant here: the first column corresponds to the load time pre-refactoring, and the middle column to the load time post-refactoring. We find statistically significant

(T-test, two-tailed, 95% confidence) reductions in initial load time in all cases, with an average speedup of 29.7%, as high as 47.5%.

The size of refactored applications is smaller in all cases, which translates to a statistically significant reduction in application start times.

RQ2: How often does the transformation introduce unwanted behavioral changes?

Since the approach presented in this chapter relies on unsound static analysis, the transformations suggested by *Lazifier* are not guaranteed to preserve application behavior. In our subject applications, *Lazifier*'s refactorings caused 15 packages to be loaded lazily, introducing 21 dynamic imports to those packages, requiring 47 other transformations (i.e., applications of a rewrite rule). We manually interacted with the applications and ensured that all transformed code was exercised, and found no behavioral differences introduced by the transformation.

For the 10 subject applications under consideration in this evaluation, there was no evidence of behavioral differences due to unsoundness in the static analysis.

RQ3: How much code is loaded lazily, and how quickly is it loaded?

When a package is loaded dynamically, the application asynchronously fetches package code and executes it, making the package available. Dynamically loading packages may result in a larger total application size, since dynamic imports load the entire package code (so no tree-shaking can be done as in the case of static imports). The total expanded size of each application is reported in column **Expanded Size (mb)** in Table 7.1. Interestingly, we note that the total size of applications after dynamic loading is always the same as the initial size without refactoring, suggesting that tree-shaking is not an effective technique at reducing the size of imported packages.

We also noted the time taken to perform this transfer, reported in Fig. 7.4, specifically the third column ("dynamic") in each set of three. The transfer is small relative to initial load times in all cases (85.8ms on average), though note that we do not simulate latency

in this test, and assuredly transferring data over a network would incur overhead related to latency.

The total size of the code loaded by the refactored applications (including lazily loaded packages) is comparable to the total size of the original applications, and dynamically loading packages is generally not noticeable.

RQ4: How many code changes are required to support lazy loading?

Since *Lazifier* suggests code changes that should be vetted carefully by programmers, it would be helpful if the extent of the transformations required was small and manageable. Table 7.2 lists information about the code transformations suggested by *Lazifier* in each subject application, namely how many packages could be loaded lazily (column **# Imps. Removed**), how many dynamic import statements were required to lazily load the packages (column **# Dyn. Imps.**), and finally how many applications of other rewrite rules were necessary to support lazily loading the packages (column **# Trans. Changes**). All cases required few code transformations, at most 15 for **upoint-query-builder** (the number of changes including added dynamic imports), with a median of 6 changes (again including added dynamic imports) per application, which should be manageable for a developer to review.

The number of code changes suggested by *Lazifier* is small, so the effort needed by programmers to review these changes is manageable.

RQ5: What is the running time of *Lazifier*?

The time taken to run *Lazifier* is reported in column **Tool Run Time (s)** of Table 7.1. This includes the time to run the static analyses and also transform the application, though the transformation itself runs extremely quickly. The time to build the CodeQL database is reported in column **QLDB Time (s)** in Table 7.1: this is a fixed cost once per project, and can be reused by other CodeQL queries.

The run time of *Lazifier* is 240s on average, demonstrating its suitability for practical use.

Table 7.2: Information about code transformations. The first row reads: *in* **upoint-query-builder**, *2 packages were loaded dynamically instead of statically; 3 dynamic import statements were added, and 12 applications of other rewrite rules were required to support the transition.*

Project Name	# Imps. Removed	# Dyn. Imps.	# Trans. Changes
upoint-query-builder	2	3	12
excelreader	1	1	2
task	1	1	2
react-excel	1	1	2
Movies-web-ui	2	2	5
ExcelSheet_Validation_Reactjs	2	3	7
scrambles-matcher	1	2	4
timetable	1	2	4
workday-schedule-exporter	3	4	6
react-excel-csv	1	2	3
In total:	15	21	47

7.6 Threats to Validity

The technique presented in this chapter was inspired by the work of Gokhale et al. [100], and suffers similar threats to validity. Namely, the code transformations proposed by our approach are unsound and are not guaranteed to preserve program behavior. There are many reasons for losses of soundness, e.g., the static analyses that build call graphs are unsound, and our technique introduces asynchrony to applications which may cause data races. In a sense, this unsoundness is inevitable as JavaScript is a highly dynamic language not amenable to sound static analysis. Nevertheless, in our evaluation we found that *Lazifier* proposed no behavior-altering transformations in spite of this unsoundness.

Beyond this, it is possible that our set of subject applications may not be representative. To mitigate this, we selected our subject applications from a list of client-side JavaScript applications sampled essentially randomly from GitHub. We did prune this list such that we could build and run the applications to evaluate the effectiveness of our technique, but believe that our random initial selection of projects mitigates risk of bias.

7.7 Relation to Previous Work

This work is concerned with refactoring web application source code to lazy load libraries that are only conditionally required. Software debloating is a related area of research focused on trimming unused functionality from applications and has many applications in security, particularly when unused code is removed from applications. Also, the refactoring

proposed in this work introduces asynchrony to an application, which is another well-studied area of research.

Debloating and Lazy Loading Chapter 3.4 discusses a wealth of work related to reducing application size. Broadly, software debloating is concerned with removing *unused* functionality, and often lazily loading the dead code if they were wrong about the code being dead, whereas the approach discussed in this chapter removes *conditionally* used functionality. In a sense, these approaches are complementary.

Refactoring to Introduce Asynchrony Loading packages lazily must be done asynchronously on the web, as blocking I/O operations are prohibited in the modern web standard. Thus, the refactoring proposed in this chapter also refactor the applications to be asynchronous w.r.t. the lazily loaded packages. There are numerous pieces of related work in this area, discussed in Chapter 3.2. Essentially, making synchronous code asynchronous is a difficult problem; in our work, we introduce *just enough* asynchronous constructs to allow for packages to be lazily loaded.

There is also a related wide body of work on *understanding* asynchronous applications, discussed in Chapter 3.3. This is complementary to our work, as *Lazifier* presents refactorings (that introduce asynchrony!) as suggestions to be vetted by programmers.

7.8 Conclusion

Client-side developers want to minimize the amount of time users need to wait for a web application to load and become responsive. Existing tools such as bundlers, minifiers, and tree-shakers focus on eliminating unused functionality and reducing code size, but do not address scenarios where an application contains functionality that is (potentially) required, but not immediately when the application starts up. In such cases, responsiveness can be improved by loading such functionality lazily. We have presented an approach for detecting situations where an entire library can be loaded lazily. The approach uses static analysis to identify packages that are only used in the context of event handling and to compute the changes that must be made to the code to accommodate lazy loading. A set of declarative rewrite rules specifies the code changes required to transform the application.

This approach was implemented in a tool called *Lazifier*, and evaluated on 10 open-source client-side JavaScript applications. In all cases, *Lazifier* successfully refactored the

applications, resulting in an average initial application size reduction of 36.2%, which caused applications to start up 29.7% more quickly on average.

7.9 Discussion

In this work, we developed an approach to detect packages that were referenced only in the context of event handlers, and developed a program transformation to load those packages lazily. This serves to reduce the size of the code that is loaded initially, i.e., the size of the *initial distribution* of an application. The approach was implemented with an unsound analysis, which did not cause issues in our evaluation (beyond possibly having missed refactoring opportunities, but significant reductions in the size of initial distributions were observed in spite of this). It appears that there are enough instances of programmers using packages *only* in event-handling code to achieve significant size reduction in applications, and an imprecise analysis could detect many such cases.

This approach is complementary to the one presented in Chapter 6, which described a method for safely removing dead code with unsound analysis. If an application has a test suite that exercises an optional dependency, dead code elimination would not remove it, even though it is only conditionally needed. If the package were only used in an event-handling context, *Lazifier* would lazily load it, achieving further size reduction on top of dead code elimination alone.

There is the possibility that behavioral differences are introduced by the code transformation, particularly in making code asynchronous. In their work on transforming uses of synchronous APIs to their asynchronous equivalents, Gokhale et al. [100] found that code changes often spanned large portions of the application and so behavioral changes *were* introduced. In contrast, in this work code changes were more localized to event handlers and functions callable from them; one advantage is that the extent of transformations was smaller, and another advantage is that event handlers are contexts that handle asynchrony by design.

If the approach described in this chapter misses a reference to a removed static import, a significant behavioral difference would be introduced. In this case, the transformation would not insert a dynamic import for the referenced module component, and if the code was run then a `ReferenceError` would occur. In many cases these "referenced before declaration" issues can be caught by the simple static analyses of linters, but nevertheless this represents a departure from the optimizations discussed in previous chapters in that inadequacies in the analysis can lead to runtime errors, rather than just missed optimization opportunities or redundant optimizations.

```
467  import React, { Suspense } from "react";
468  import { DogPage } from "./DogPage.js";
469  import { CatPage } from "./CatPage.js";
470
471  export default (props) => {
472    if (props.user.selection === "dog") {
473      return <DogPage />;
474    } else if (props.user.selection === "cat") {
475      return <CatPage />;
476    } else {
477      return <div> Only dog and cat adoptions supported online. </div>;
478    }
479  };
```

Figure 7.5: Example pet adoption service application.

One interesting avenue of future work is to lazily load entire components of the UI, as entire application does not need to be loaded for one page of the UI to be functional. This is a more complicated transformation that also requires placeholder UI elements to be displayed while components load dynamically, and unfortunately the program transformation differs depending on the UI framework being used (in JavaScript, most client-side applications are built with a UI framework), as each UI frameworks has its own mechanism for loading UI elements lazily. For example, React has a `lazy` function [150] and functionality for displaying placeholder components while sub-components load [151]. Unlike React, in Angular programmers must create lazily loaded routes using `loadChildren` [46]; the Angular bundler takes advantage of these to split the application bundle.

To get a flavor for what this would look like, consider a hypothetical multi-page React application for an animal adoption service. Such an application might first ask a user what kind of animal they want to adopt, and display a subsequent page depending on that selection. Some example code can be found in Figure 7.5. This code displays the UI once a user has indicated which kind of animal they want to adopt. On line 468 the UI elements related to dog adoption are imported, and on line 469 the UI elements for cat adoption are imported. These are JSX components, and they are displayed to the user on lines 473 and 475 depending on the user's selection, with an else branch indicating that only cat and dog adoptions are supported on the site. The issue in the snippet is that all of the functionality is loaded when the application starts, which is very wasteful.

Figure 7.6 shows how the application can be refactored to load components lazily. To lazily load a component in React, it needs to be imported dynamically using `import` (like the approach described in this chapter), and then wrapped in a call to React's `lazy` function; we see this on lines 481-482. Then, references to the module need to be wrapped in a `<Suspend>` component, which takes a suitable placeholder UI element called a *fallback* to be

```
480  import React, { Suspense } from "react";
481  const DogPage = React.lazy(() => import("./DogPage.js"));
482  const CatPage = React.lazy(() => import("./CatPage.js"));
483
484  export default (props) => {
485    if (props.user.selection === "dog") {
486      return (
487        <Suspense fallback={<div>Loading dog adoption service...</div>}>
488          <DogPage />
489        </Suspense>
490      );
491    } else if (props.user.selection === "cat") {
492      return (
493        <Suspense fallback={<div>Loading cat adoption service...</div>}>
494          <CatPage />
495        </Suspense>
496      );
497    } else {
498      return <div> Only dog and cat adoptions supported online. </div>;
499    }
500  };
```

Figure 7.6: Example pet adoption service application, now with lazy loading of optional UI components.

displayed while the lazy load completes; we see this on lines 486-490 and lines 492-496.

In this case, the fallback is merely some text saying that the requested adoption service is being loaded, but the differences between that and the actual UI might be glaring. It would be interesting to use some lightweight static analysis to determine the structure of a placeholder UI element that roughly matches the element that would be loaded dynamically. In many cases, the structure of UI elements is evident from the code as many of the JavaScript frameworks mimic the look of HTML (React's JSX looks like HTML, and Angular and Vue both use HTML explicitly).

Chapter 8

Conclusion

In this thesis, we set out to show that *unsound analysis of asynchronous JavaScript applications yields actionable insights and effective optimizations*. We applied unsound analysis in four settings, and found promising results in all cases.

In Chapter 4, we developed a technique for detecting anti-patterns in asynchronous JavaScript applications, hinging on lightweight unsound static analysis to detect anti-patterns and a dynamic analysis collecting information about runtime promises. This technique was implemented in a tool called *DrAsync*, and we evaluated it on 20 popular open-source JavaScript applications, finding thousands of instances of anti-patterns in them. We conducted case studies of 80 instances of the anti-patterns, and found that the vast majority could be refactored by outsiders to the code base, suggesting that the insights delivered by the approach are indeed *actionable*.

An unsound analysis is appropriate here for a few reasons: (1) the anti-patterns are simple, and imprecise analysis can detect them easily; (2) we envision *DrAsync*'s anti-pattern detector to be run essentially like a linter, and so run time should be manageable; and (3) a dynamic analysis corroborates the information gleaned from static analysis, equipping developers with more information to assist them in remediating anti-patterns, helping to make up for the lack of precision of the unsound static analysis. Unsoundness in this context means that we might miss actual anti-patterns (false negatives), and may also incorrectly flag correct code as exhibiting anti-patterns (false positives), though we did not observe much of this empirically.

In Chapter 5, we presented an approach for improving the performance of database-backed applications via automated refactoring. An unsound static taint analysis tracks data flow from ORM API calls through a loop into other ORM API calls, identifying pairs

of such data-related calls as instances of the "N+1 Problem", and declarative rewrite rules specify how such pairs of calls should be transformed to eliminate the "N+1 Problem" altogether. This technique was implemented in a tool called REFORMULATOR, which we evaluated on 8 open-source JavaScript applications. We found many instances of the "N+1 Problem" and found that REFORMULATOR was able to successfully refactor all instances, resulting in significant performance improvements in the applications, representing an *effective optimization* of the program.

Unlike in Chapter 4, here we were able to automatically determine the code changes required to remediate the issue. Even though we still use imprecise and unsound analysis, there are many data-related ORM API calls that co-occur in relatively close proximity in the code. Further, these API calls have very strict requirements on their arguments and predictable return types, which seems to discourage programmers from writing very dynamic code when preparing or using the values obtained from API calls and provides information that imprecise analysis can take advantage of. Taken together, this suggests that the *overall* lack of precision of our approach does not lead to a significant lack of precision *in this case*; i.e., while we do not have complete information, the information we have is good enough for this context. Of course, code transformation opportunities are still detected through unsound analysis, and the rewrite rules are not sound. The approach might miss refactoring opportunities (false negatives), might incorrectly flag pairs of ORM API calls for refactoring (false positives), and the transformations themselves may not preserve program behavior; that said, we found no issues that arose due to unsoundness in practice.

In Chapter 6, we rephrased dead code elimination to instead aggressively apply code splitting to likely dead code, leveraging imprecise unsound analysis. Unsound program analysis (static or dynamic) builds a call graph for an application, and unreachable functions and files are replaced with stub versions that can fetch the original code if the call graph was wrong about the code being dead. This allows for significant size reduction; we implemented this technique in a tool called *Stubbifier*, and ran it on 20 open-source JavaScript applications, finding 56% initial application size reduction on average, and that little code was fetched dynamically incurring manageable overhead. Programmers want small distributions, and this tool and technique help them achieve that with another *effective optimization*.

Unsound analysis was successful in this setting because the consequences of wrongfully applying the code transformation were not catastrophic: if the code was not dead, then it is loaded dynamically. We took a program optimization that only made sense with a sound and precise static analysis, and re-framed it to work with unsound and imprecise analysis. The fact that sound analysis vastly over-approximates reachability in JavaScript was key

to the success of this approach was. Unsoundness in the analysis can result in code being incorrectly identified as live (false negative) and code being incorrectly identified as dead (false positive); unsoundness in the transformations may result in behavioral differences introduced by the code changes. In the evaluation, we found some false positives but the code transformation was explicitly designed to handle this case, and no behavioral differences were observed empirically (i.e., no refactorings caused tests to fail).

Finally, in Chapter 7, we developed an approach for automatically lazily loading packages in client-side JavaScript applications. A lightweight, imprecise static analysis determines what is transitively callable from event-handling code, and any package *exclusively* used in such contexts is flagged to be loaded lazily, and declarative rewrite rules specify how the code should be transformed to lazily load the package. We implemented this approach in a tool called *Lazifier*, and evaluated it on 10 open-source JavaScript applications, finding that *Lazifier* successfully lazily loaded many packages, and that initial code size was significantly reduced in all cases. This is a boon for client-side developers, who desire lean initial application sizes; as such, this approach is yet another *effective optimization* for client-side programs.

Unsoundness in this context means that refactoring opportunities might be missed, and that the suggested transformations may not preserve application behavior. This setting is similar to the one explored in Chapter 6, where the consequences of loading a package lazily that was needed eagerly are not severe: if a package was actually needed on startup, it will simply be loaded dynamically when the application page is first visited, and missed refactoring opportunities do not cause errors in the program (these are not bugs, only inefficiencies). As for transformations possibly not preserving application behavior, this is unavoidable given the unsound analysis and program transformations (as in Chapter 5), but in this case the risk is mitigated by having the analysis focus on event-handling code, which minimizes the risk of incorrectly transforming code since the suggested refactorings tend to span a small part of the application.

8.1 Discussion

In each setting, we had to account for imprecision and unsoundness of the analyses. In this section, we reflect on this and discuss the benefits and limitations of employing unsound methods to optimize programs.

8.1.1 Dynamic vs. Static Analysis

Static analysis analyzed a program without running it, and instead models the behavior of the program; the quality of the model has a huge impact on the precision and correctness of the results, and on the scalability of the analysis. In contrast, dynamic analysis analyzes a program while it is running; the gathering of information at runtime has a performance impact, and dynamic analysis is typically quite precise in that an actual program execution is observed. That said, it is not always obvious how analysis results from one execution generalize to all executions, while static analysis explores possible program behavior more broadly. In this thesis we developed many approaches relying primarily on static, rather than dynamic program analysis, but in many cases we initially explored approaches using dynamic analysis.

In Chapter 4 (*DrAsync*), the initial approach relied solely on dynamic analysis. We built the promise profiler and accompanying visualization and pored over hundreds of JavaScript projects (with running test suites, as they are needed for the dynamic analysis). We identified many issues just from looking at the dynamic information, chiefly extremely short-lived promises and the "staircase pattern" discussed in Chapter 4.

In Chapter 6 (*Stubbifier*), the code splitting approach takes a call graph as input, which can be computed statically or dynamically, and the chapter discusses and evaluates call graphs obtained using both methods. In the tool, the (`nyc`) coverage reporter was used to obtain a dynamic call graph with respect to the execution of the test suite. Similarly, a simple static analysis used the test suite as an entry point to determine what is reachable.

In Chapter 5 (REFORMULATOR), we originally used a dynamic taint analysis built in Augur [39]; Augur is a taint analysis framework that leverages the NodeProf [204] dynamic analysis framework for GraalVM [237, 235]. As discussed in Section 5.9, the precise data flow gleaned with dynamic analysis was not necessary for the code transformations that fixed instances of the "N+1 Problem". The precision of a dynamic analysis could be leveraged by more sophisticated transformations.

Dynamic analysis has a very important drawback: the code needs to be (able to be) executed, and this turns out to be extremely challenging. In Chapter 4, the projects had test suites that executed, so this was not an issue, but the vast majority of JavaScript projects on GitHub have no running test suites. It is difficult to apply an approach that relies too heavily on dynamic analysis given the current developer culture around testing, and in terms of research it is difficult to *evaluate* approaches that rely heavily on dynamic analysis due to the lack of easily executable code. This can be alleviated with test generation, though test generation as it applies to dynamic languages has its own set of challenges [218, 47, 193], mainly due to the lack of effective implicit oracles.

Another important limitation of dynamic analyses is that they are rarely portable. There are two predominant approaches to obtaining dynamic information from programs: instrumentation, and source code rewriting. Instrumentation is difficult because it requires modifying the underlying virtual machine to emit dynamic information, which is not portable (unless V8 were to expose dynamic analysis hooks, in which case the lion's share of JavaScript could be analyzed!). Source code rewriting is more portable, but significantly degrades the performance of analyzed applications. Moreover, JavaScript is extremely dynamic and reflective, and dynamic analyses that modify runtime objects can break applications if programmers reflect on the shape of objects. In the *DrAsync* tool, we implemented the dynamic analysis with the `Async Hooks` API exposed by Node.js, and *a lot* of JavaScript runs on Node.js. In *Stubbifier*, the dynamic call graph was obtained using a coverage reporter, which is very portable and used broadly by the community. The initial analysis underpinning REFORMULATOR was built in Augur (built in NodeProf and GraalVM), and we had difficulty running many JavaScript projects in GraalVM.

There is a wealth of related work on combining static and dynamic analysis information, discussed in Section 3.1.3. In the refactoring projects presented in Chapters 5 and 7 we decided against this due to limitations in the portability of dynamic analysis, and further the added precision is likely unnecessary. In Chapter 4, where dynamic and static analysis feature, we felt the added precision would not have helped detect any of the anti-patterns proposed in the work, and instead present information from both sources to programmers for them to make sense of. In Chapter 6, we investigated totally removing functions deemed unreachable in *both* static and dynamic analysis, but this caused a lot of live code to be removed.

Takeaway: Dynamic analysis yields precise information, but generalizing beyond a particular execution is difficult. Moreover, analyzing a program dynamically incurs oftentimes significant overhead, and dynamic analyses are less portable than static analyses.

8.1.2 Empowering Programmers

As mentioned, *DrAsync* was originally envisioned as an entirely dynamic promise profiling tool. One reason we shifted away from a pure dynamic analysis approach was because we believed that our familiarity with the JavaScript semantics and runtime was helping us understand the dynamic profile information, and that the profiles might not be as easily understood by developers. Besides, there were common code patterns that resulted in the problematic execution profiles, and these patterns had enough consistency that we were able to specify them formally, and devise simple static analyses to detect them. Shifting

to detection of the code patterns resulting in the issues in the dynamic profiles was a way to leverage our expertise to communicate the most salient details to programmers.

We employed an unsound and imprecise analysis to find anti-patterns, and ultimately decided to keep both the dynamic and static analyses since the *precise* information gleaned via dynamic analysis complemented the relatively imprecise information obtained from static analysis. We felt that the dynamic profile information was even *more* useful with a static analysis highlighting anti-patterns since it makes the impact of the anti-patterns on program execution clear. The complementary nature of the analyses was made explicit by connecting anti-pattern code to the runtime promises it created in the visualization. Another important feature of the dynamic analysis is establishing the relationships between runtime promises by tracing promise chains, very *precise* information about promises.

The anti-pattern identification part of *DrAsync* is similar in spirit to linters, although it can be thought of as more of a linting suite for promise-related issues, complete with dynamic information to help programmers determine the impact of issues and prioritize fixing them. It is really a tool to empower programmers, equipping them with information to modify their code by showing them anti-patterns in their code and connecting those with their impact on test suite execution.

In Chapters 5- 7, program transformations to optimize code are suggested to programmers, which is an important distinction from transformations enacted by compilers and language runtimes to optimize code as it is compiled or run. In addressing inefficiencies at their source, this thesis describes how inefficiently written source code can be detected and, in many cases, fixed automatically ahead-of-time via automated refactoring, and as programmers review the transformations they are made aware of the issues in the code they wrote.

In Chapter 5 we proposed an approach for automatically detecting and remediating instances of the "N+1 Problem" in ORM-backed web applications. The underlying issue that the approach detected was the data flow between ORM API calls that passed through a loop, and in Section 5.9 we discussed future work where *all* data flow between ORM API calls should be called into question. When a developer uses REFORMULATOR, the tool will suggest code transformations, but can additionally serve to educate the developer that they should, e.g., avoid data flow between ORM API calls in general. In contrast, tools that optimize inefficient database use in the back-end provide no insight to programmers, and they will continue to write the same inefficient code patterns.

Similarly in Chapter 6, the approach removes code based on what is executable from the test suite, and if the programmer is informed about what this code is they have information that would help them improve the test suite. In Chapter 7, only packages that are used

exclusively in the context of event handlers are loaded lazily; in suggesting a refactoring, programmers are provided an example of how to load packages lazily, which they could learn from and apply throughout their code.

There is rich avenue for future work in incorporating program understanding techniques in the approaches presented in this thesis, or more broadly in the space of refactoring suggestions. In Chapters 5 and 7, we determined refactorings and suggested them to programmers—when programmers are deciding whether or not to apply a refactoring, future work might explore how to best equip them to accept or reject the transformation. We focused the evaluations on exploring if the transformation worked, rather than if programmers want to apply it. There are often further considerations beyond raw performance or size reduction. E.g., in Chapter 5, eagerly fetching data from a database can exert a high degree of pressure on application memory. That said, we found no issues related to this in the evaluation presented in the context of the chapter.

Takeaway: Imperfect, unsound analysis extracts useful information for programmers. Multiple sources of information further empowers programmers to understand and make changes to their code. Moreover, suggesting optimizations ahead of time gives programmers insight into the issues they are responsible for.

8.1.3 Finding Precision Where You Can

Many approaches presented in this thesis rely on precise information about specific parts of programs. In Chapter 4, precise information about runtime promises helps corroborate general information obtained through static analysis. In Chapter 5, precise data flow maps pairs of N+1-related ORM API calls, and precise information about object properties are needed to build code transformations. In Chapter 7, a precise call graph of event-handling code was instrumental in building and validating code transformations to lazily load packages. In this section, we reflect on how such precise information was sourced in these contexts.

Dynamic Analysis

This was discussed in § 8.1.1, but to recap dynamic analysis is a source of very precise information about particular executions of a program. Generalizing information beyond any given execution is challenging, and generally there are obstacles and limitations to running a dynamic analysis (including having sufficient executable code, the portability of the analysis, and significant performance overhead associated with dynamic analysis).

Analysis developers can opt to incorporate dynamic information directly into a static analysis, or vice versa. We did not do that in the work presented in this thesis as the approaches worked well without the added precision. § 3.1.3 discusses related literature.

Strict APIs

Many API boundaries have strict requirements that need to be met. This strictness is advantageous to analyses, as the arguments supplied to and values obtained from calls to these APIs are often predictable.

We observed this while developing the approach presented in Chapter 5 (REFORMULATOR). The ORM API takes an object with property names that are either from a set of pre-defined options, or correspond to columns in the database the ORM is interacting with; and ORM API calls typically return arrays of objects with properties drawn from the statically available model of the tables in the database. Thus, the types of arguments and return values are known to even imprecise analyses.

Further, in Chapter 7 (*Lazifier*) the strict, static nature of ECMAScript 2015's `import` made references to exported properties of external packages straightforward to detect. With static `import`, programmers must supply the name of the external package as a string literal, so determining which packages were imported was simple. In contrast, the older import functionality using `require` allowed programmers to supply dynamically computed strings, which would have been more demanding for a static analysis.

Heuristics

While the approaches presented in the context of this thesis were principled in their overall design, there were several situations where heuristics greatly enhanced the quality of the results.

For example, in Chapter 7 we developed a method to automatically lazily load packages that relied on identifying packages that were used exclusively in the context of event-handling code. In JavaScript, there is no single mechanism for "handling events"; HTML elements have attributes with no consistent naming scheme that specify event handlers, programmers can register code blocks as "click" attributes as well as functions, some event handling is done with higher level abstractions (e.g., a file reader class), and functions can be directly assigned to the DOM. Also, different UI frameworks provide additional abstractions for handling events that do not correspond to any aforementioned methods. In the end, all of these methods likely result in a callback being registered for an event. A

precise analysis that delved into the abstractions to uncover callback registrations would be incredibly costly, so instead these heuristics serve as *shortcuts* that allow imprecise analyses to make judgements which might otherwise be too costly.

In Chapter 5, ORM API calls in a loop are refactored into a single eager call placed before the loop. This is unsound, and is predicated on an assumption that loop iterations are independent w.r.t. the ORM API call. A (more) sound analysis should analyze the loop and confirm this assumption, because the code transformation should not be applied if the assumption is invalid, or further transformations may be required. In this case, suggesting transformations to programmers circumvents this source of unsoundness, as issues with the transformation would quickly be revealed, and programmers may be able to determine the rest of the transformation required. That said, this issue did not manifest in our evaluation.

Takeaway: Unsound should not mean unprincipled. There are very good reasons to employ unsound approaches, especially in dynamic languages where soundness and precision are often antonymous.

8.2 Closing Thoughts

In this thesis, we used unsound analysis to optimize asynchronous JavaScript programs. Specifically, we demonstrated that *unsound analysis of asynchronous JavaScript applications yields actionable insights and effective optimizations* by developing four approaches that fit into this statement. Promising results in each approach indicate that lightweight, unsound analysis can be leveraged to make meaningful, impactful, and actionable suggestions to programmers.

That said, unsound analysis is far from a silver bullet. Unsound approaches are inherently less reliable than sound approaches and lack a solid bedrock of correctness, but clever design that *mitigates* unsoundness can make up for this. In this thesis, we re-framed optimizations to be safely applied given unreliable information, we developed techniques that leverage precise and readily accessible information to build complex code transformations, and collected and distilled information and presented holistic insights it to programmers. These are imperfect methods, but the *sound* analysis that would invariably be required for sound methods is infeasible in languages as dynamic as JavaScript; here, we show that perfect is the enemy of good.

There is ample opportunity to build on the work laid out in this thesis. Essentially, we present "linting" tools that use relatively sophisticated analysis to improve the quality of

source code. There is deep literature on *program understanding*, which can help improve the delivery of refactoring suggestions. Further, one can leverage advancements in *test generation* to equip refactoring suggestions with focused code snippets to help programmers explore the execution profile and behavior of potential code changes. And of course, *program analysis* underpins all of the work discussed here, and any advancements in the field will improve the quality of results.

References

[1] Concurrency visualizer - visual studio (windows) — microsoft docs. https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019, 2017. (Accessed on 08/20/2021).

[2] Jshint: A static code analysis tool for JavaScript, 2019. See https://jshint.com/.

[3] Async hooks — node.js v16.6.0 documentation. https://nodejs.org/api/async_hooks.html, 2020. (Accessed on 08/02/2021).

[4] bdistin/fs-nextra. https://github.com/bdistin/fs-nextra, 2021. Accessed: 2021-10-25.

[5] Codeql for research — github security lab. https://securitylab.github.com/tools/codeql/, 2021. (Accessed on 08/10/2021).

[6] Commit: remove eval. https://github.com/dougwilson/nodejs-depd/commit/887283b4, 2021. Accessed: 2021-04-16.

[7] depd issue 20. https://github.com/dougwilson/nodejs-depd/issues/20, 2021. Accessed: 2021-04-16.

[8] depd issue 22. https://github.com/dougwilson/nodejs-depd/issues/22, 2021. Accessed: 2021-04-16.

[9] depd issue 24. https://github.com/dougwilson/nodejs-depd/issues/24, 2021. Accessed: 2021-04-16.

[10] dougwilson/nodejs-depd. https://github.com/dougwilson/nodejs-depd, 2021. Accessed: 2021-04-16.

[11] ECMAScript 2021 Language Specification Section 27.2: Promises. https://262.ecma-international.org/#sec-promise-objects, June 2021.

[12] Eslint: Find and fix problems in your JavaScript code, 2021. See https://eslint.org/.

[13] expressjs/body-parser. https://github.com/expressjs/body-parser, 2021. Accessed: 2021-10-25.

[14] expressjs/compression. https://github.com/expressjs/compression, 2021. Accessed: 2021-10-25.

[15] expressjs/morgan. https://github.com/expressjs/morgan, 2021. Accessed: 2021-10-25.

[16] expressjs/serve-favicon. https://github.com/expressjs/serve-favicon, 2021. Accessed: 2021-10-25.

[17] expressjs/serve-static. https://github.com/expressjs/serve-static, 2021. Accessed: 2021-10-25.

[18] facebook/prop-types. https://github.com/facebook/prop-types, 2021. Accessed: 2021-10-25.

[19] isaacs/node-glob. https://github.com/isaacs/node-glob, 2021. Accessed: 2021-10-25.

[20] Jslint, 2021. See https://www.jslint.com/.

[21] kriskowal/q. https://github.com/kriskowal/q, 2021. Accessed: 2021-10-25.

[22] mapbox/node-blend. https://github.com/mapbox/node-blend, 2021. Accessed: 2021-04-16.

[23] pillarjs/send. https://github.com/pillarjs/send, 2021. Accessed: 2021-10-25.

[24] streamich/memfs. https://github.com/streamich/memfs, 2021. Accessed: 2021-10-25.

[25] Thread concurrency visualization — pycharm. https://www.jetbrains.com/help/pycharm/thread-concurrency-visualization.html, 2021. (Accessed on 08/20/2021).

[26] Threading analysis. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/parallelism-analysis-group/threading-analysis.html, 2021. (Accessed on 08/20/2021).

[27] tj/commander.js. https://github.com/tj/commander.js, 2021. Accessed: 2021-10-25.

[28] webpack-contrib/css-loader. https://github.com/webpack-contrib/css-loader, 2021. Accessed: 2021-10-25.

[29] webpack/memory-fs. https://github.com/webpack/memory-fs, 2021. Accessed: 2021-10-25.

[30] Laravel: The PHP framework for web artisans, 2022. See https://laravel.com/.

[31] Sequelize ORM, 2022. See https://sequelize.org.

[32] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), November 2009.

[33] Abhishek312s. Movies-web-ui, 2023. See https://github.com/Abhishek312s/Movies-web-ui/58904a3.

[34] adam dill. wall, 2022. See https://github.com/adam-dill/wall/commit/ae6c815.

[35] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. Practical initialization race detection for JavaScript web applications. *Proc. ACM Program. Lang.*, 1(OOPSLA):66:1–66:22, 2017.

[36] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, pages 247–267, 1993.

[37] Ole Agesen and David Ungar. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 355–370, Portland, OR, 1994. *ACM SIGPLAN Notices* 29(10).

[38] Akalay27. workday-schedule-exporter, 2023. See https://github.com/Akalay27/workday-schedule-exporter/97ca596.

[39] Mark W Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. Augur: Dynamic taint analysis for asynchronous javascript. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[40] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid dom-sensitive change impact analysis for javascript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[41] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack javascript. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1169–1180, 2016.

[42] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding javascript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 367–377, 2014.

[43] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proc. ACM Program. Lang.*, 2(OOPSLA):162:1–162:26, 2018.

[44] Scott Ambler and Pramod Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley, 1 edition, 2006.

[45] Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proc. 29th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA)*, 2014.

[46] Angular. Angular - LoadChildrenCallback, 2023. See https://angular.io/api/router/LoadChildrenCallback.

[47] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. Nessie: Automatically testing javascript apis with asynchronous callbacks. In *Proceedings of the International Conference on Software Engineering (ICSE 2022)*, 2022.

[48] Ellen Arteca, Max Schäfer, and Frank Tip. Learning how to listen: Automatically finding bug patterns in event-driven JavaScript APIs. *IEEE Trans. Software Eng.*, 2022.

[49] Ellen Arteca, Frank Tip, and Max Schäfer. Enabling additional parallelism in asynchronous JavaScript applications. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 7:1–7:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[50] Ellen Arteca and Alexi Turcotte. Npm-filter: Automating the mining of dynamic information from npm packages. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 304–308, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 2:1–2:25, 2016.

[52] Babel. Babel, 2022. See https://babeljs.io/.

[53] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996.*, pages 324–341, 1996.

[54] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, 2008.

[55] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. Combining concern input with program analysis for bloat detection. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 745–764, New York, NY, USA, 2013. ACM.

[56] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. JShrink: In-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 135–146, 2020.

[57] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 439–453. USENIX Association, 2015.

[58] Boyuan Chen, Zhen Ming Jiang, Paul Matos, and Michael Lacaria. An industrial experience report on performance-aware refactoring on a database-centric web application. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 653–664. IEEE, 2019.

[59] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1001–1012, New York, NY, USA, 2014. Association for Computing Machinery.

[60] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, 42(12):1148–1161, 2016.

[61] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 931–942. ACM, 2014.

[62] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, 2010.

[63] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[64] Douglas Crockford. jsmin, 2023. See https://www.crockford.com/jsmin.html.

[65] daedadev. employee-tracker, 2022. See https://github.com/daedadev/employee-tracker/commit/ba4a195.

[66] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Sifting out the mud: Low level C++ code reuse. *SIGPLAN Notices*, 37(11):275–291, November 2002.

[67] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, pages 77–101, 1995.

[68] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: are we there yet? In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pages 612–621. IEEE, 2018.

[69] Danny Dig. A refactoring approach to parallelism. *IEEE Softw.*, 28(1):17–22, 2011.

[70] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 397–407. IEEE, 2009.

[71] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph E. Johnson. Relooper: refactoring for loop parallelism in java. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 793–794. ACM, 2009.

[72] Sourav Dutta, Sheheeda Manakkadu, and Dimitri Kagaris. Classifying performance bottlenecks in multi-threaded applications. In *IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC 2014, Aizu-Wakamatsu, Japan, September 23-25, 2014*, pages 341–345. IEEE Computer Society, 2014.

[73] ECMA. Ecmascript 2021 language specification, 2021. Available from https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[74] ECMA International. ECMAScript 2019 language specification. https://262.ecma-international.org/10.0/, 2019. Accessed: 2021-04-16.

[75] ECMA International. ECMAScript module system. https://www.ecma-international.org/ecma-262/#sec-modules, 2021. Accessed: 2021-04-16.

[76] Elektra-GHP. Graceshopper-elektra, 2022. See https://github.com/Elektra-GHP/Graceshopper-Elektra/commit/c327530.

[77] eligrey. file-saver, 2023. See https://www.npmjs.com/package/file-saver.

[78] employee tracker, 2022. See https://github.com/daedadev/employee-tracker/blob/main/index.js#L9-L44.

[79] employee tracker, 2022. See https://github.com/daedadev/employee-tracker/blob/main/index.js#L169-L219.

[80] eventbright, 2022. See https://github.com/twincarlos/eventbright/blob/main/backend/routes/api/events.js#L17-L31.

[81] eventbright, 2022. See https://github.com/twincarlos/eventbright/blob/main/backend/routes/api/events.js#L32-L43.

[82] eventbright, 2022. See https://github.com/twincarlos/eventbright/blob/main/backend/routes/api/events.js#L44-L63.

[83] eventbright, 2022. See https://github.com/twincarlos/eventbright/blob/main/backend/routes/api/events.js#L64-L76.

[84] eventbright, 2022. See https://github.com/twincarlos/eventbright/blob/main/backend/routes/api/events.js#L104-L114.

[85] eventbright, 2022. See https://github.com/twincarlos/eventbright/blob/main/backend/routes/api/like.js#L6-L16.

[86] eventbright, 2022. See https://github.com/twincarlos/eventbright/blob/main/backend/routes/api/order.js#L6-L21.

[87] fahimahammed. task, 2023. See https://github.com/fahimahammed/task/b641bc0.

[88] Amin Milani Fard and Ali Mesbah. JSNOSE: detecting javascript code smells. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, pages 116–125. IEEE Computer Society, 2013.

[89] Asger Feldthaus, Todd D. Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for JavaScript. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 119–138. ACM, 2011.

[90] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761, 2013.

[91] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1 edition, 1999.

[92] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[93] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 2 edition, 2018.

[94] Dennis F. Galletta, Raymond M. Henry, Scott McCoy, and Peter Polak. Web site delays: How tolerant are users? *J. Assoc. Inf. Syst.*, 5(1):1, 2004.

[95] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. A¡span class=”smallcaps smallercapital”¿ffogato¡/span¿: Runtime detection of injection attacks for node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA ’18, page 94–99, New York, NY, USA, 2018. Association for Computing Machinery.

[96] GitHub. Language trends on GitHub. https://octoverse.github.com/#top-languages, 2020.

[97] GitHub. CodeQL. https://github.com/github/codeql, 2021. Accessed: 2021-04-16.

[98] GitHub. State of the Octoverse, 2023. See https://octoverse.github.com/2022/top-programming-languages.

[99] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, jun 2005.

[100] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[101] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. Dlint: dynamically checking bad coding practices in javascript. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 94–105. ACM, 2015.

[102] Google. Chrome DevTools, 2022. See https://developer.chrome.com/docs/devtools/.

[103] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. A true positives theorem for a static race detector. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[104] Graceshopper-Elektra, 2022. See https://github.com/Elektra-GHP/Graceshopper-Elektra/blob/master/server/api/checkout.js#L7-L47.

[105] Neville Grech and Yannis Smaragdakis. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[106] Maila Hardin, Daniel Hom, Ross Perez, and Lori Williams. Which chart or graph is right for you? *Tell Impactful Stories with Data. Tableau Software*, 2012.

[107] Harinathlee. upoint-query-builder, 2023. See https://github.com/Harinathlee/upoint-query-builder/f9aa0f1.

[108] Hibernate. What is object/relational mapping?, 2022. See http://hibernate.org/orm/what-is-an-orm/.

[109] hongtaodai. react-excel, 2023. See https://github.com/hongtaodai/react-excel/2d59e85.

[110] David Hovemeyer and William Pugh. More efficient network class loading through bundling. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, pages 127–140, 2001.

[111] hoverGecko. timetable, 2023. See https://github.com/hoverGecko/timetable/0fa8527.

[112] IBM Corporation. *VisualAge for Smalltalk Handbook Volume 1: Fundamentals*, first edition edition, 1997. Available from http://www.redbooks.ibm.com/redbooks/4instantiations/sg244828.pdf.

[113] Istanbul. nyc. https://www.npmjs.com/package/nyc, 2021. Accessed: 2021-10-12.

[114] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 34–44. ACM, 2012.

[115] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 59–69, 2011.

[116] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[117] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 223–234, 2012.

[118] David Johannes, Foutse Khomh, and Giuliano Antoniol. A large-scale empirical study of code smells in JavaScript projects. *Softw. Qual. J.*, 27(3):1271–1314, 2019.

[119] Wagner Meira Jr., Thomas J. LeBlanc, and Alexandros Poulos. Waiting time analysis and performance visualization in carnival. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT'96)*, pages 1–10, 1996.

[120] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[121] Rezwana Karim, Frank Tip, Alena Sochurková, and Koushik Sen. Platform-independent dynamic taint analysis for JavaScript. *IEEE Trans. Software Eng.*, 46(12):1364–1379, 2020.

[122] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 121–132, New York, NY, USA, 2014. Association for Computing Machinery.

[123] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering*, 40(9):841–861, 2014.

[124] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. Safe automated refactoring for intelligent parallelization of Java 8 streams. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 619–630. IEEE / ACM, 2019.

[125] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. DAPP: automatic detection and analysis of prototype pollution vulnerability in node.js modules. *Int. J. Inf. Sec.*, 21(1):1–23, 2022.

[126] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the Attack Surface of Node.js Applications. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, October 2020.

[127] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.

[128] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Dynamic taint analysis with label-defined semantics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, MPLR '22, page 64–84, New York, NY, USA, 2022. Association for Computing Machinery.

[129] Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing transfer delay using Java class file splitting and prefetching. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, pages 276–291, 1999.

[130] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In Diomidis Spinellis, Georgios

Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 268–279. ACM, 2021.

[131] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, 2022. USENIX Association.

[132] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *PACMPL*, 2(OOPSLA):141:1–141:29, 2018.

[133] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 129–140, 2018.

[134] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for Android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 341–352, 2014.

[135] Gitte Lindgaard, Gary Fernandes, Cathy Dudek, and Judith M. Brown. Attention web designers: You have 50 milliseconds to make a good first impression! *Behav. Inf. Technol.*, 25(2):115–126, 2006.

[136] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 422–433, 2015.

[137] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014.

[138] V. Benjamin Livshits and Emre Kiciman. Doloto: code splitting for network-bound Web 2.0 applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 350–360, 2008.

[139] Yingjun Lyu, Ding Li, and William G. J. Halfond. Remove rats from your code: Automated optimization of resource inefficient database writes for mobile applications.

In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 310–321. ACM, 2018.

[140] Magnus Madsen, Ondrej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proc. ACM Program. Lang.*, 1(OOPSLA):86:1–86:24, 2017.

[141] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 499–509, New York, NY, USA, 2013. Association for Computing Machinery.

[142] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven node.js javascript applications. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 505–519. ACM, 2015.

[143] Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, Simone Romano, Salvatore Lombardi, Giuseppe Scanniello, and Patricia Lago. Javascript dead code identification, elimination, and empirical assessment. *IEEE Transactions on Software Engineering*, pages 1–23, 2023.

[144] manikandanraji. youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/commit/47002fc.

[145] Math_Fluency_App, 2022. See https://github.com/rayace5/Math_Fluency_App/blob/main/routes/results.js#L428-L511.

[146] Math_Fluency_App, 2022. See https://github.com/rayace5/Math_Fluency_App/blob/main/routes/results.js#L603-L686.

[147] Math_Fluency_App, 2022. See https://github.com/rayace5/Math_Fluency_App/blob/main/routes/results.js#L259-L336.

[148] Lauren McCarthy, Casey Reas, and Ben Fry. *Getting started with P5. js: Making interactive graphics in JavaScript and processing*. Maker Media, Inc., 2015.

[149] MDN. Tree shaking. https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking, 2021. Accessed: 2021-10-11.

[150] Meta. lazy - react, 2023. See https://react.dev/reference/react/lazy.

[151] Meta. Suspense - react, 2023. See https://react.dev/reference/react/Suspense.

[152] Microsoft. CodeQL, 2022. See https://codeql.github.com/.

[153] Microsoft. CodeQL, 2022. See https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-javascript-and-typescript/#analyzing-data-flow-in-javascript-and-typescript.

[154] Microsoft. CodeQL JavaScript data flow library, 2023. See https://github.com/github/codeql/tree/7323d4e/javascript/ql/lib/semmle/javascript/dataflow.

[155] mikethecodegeek. property-manage, 2022. See https://github.com/mikethecodegeek/property-manage/commit/33f92a9.

[156] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[157] mishoo. uglify-js, 2023. See https://www.npmjs.com/package/uglify-js.

[158] Anders Møller. Jelly, 2023. See https://github.com/cs-au-dk/jelly.

[159] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in javascript programs affected by breaking library changes. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[160] Anders Møller and Martin Toldam Torp. Model-based testing of breaking changes in node.js libraries. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 409–419, New York, NY, USA, 2019. Association for Computing Machinery.

[161] Mozilla. Rest parameters. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters, 2021. Accessed 2021-04-16.

[162] NetSteam, 2022. See https://github.com/W-the-V/NetSteam/blob/main/backend/routes/api/reviews.js#L14-L42.

[163] NetSteam, 2022. See https://github.com/W-the-V/NetSteam/blob/main/backend/routes/api/reviews.js#L44-L80.

[164] NetSteam, 2022. See https://github.com/W-the-V/NetSteam/blob/main/backend/routes/api/reviews.js#L82-L120.

[165] NetSteam, 2022. See https://github.com/W-the-V/NetSteam/blob/main/backend/routes/api/reviews.js#L122-L157.

[166] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Detection of embedded code smells in dynamic web applications. In Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors, *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 282–285. ACM, 2012.

[167] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: Feedback-driven static analysis of node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 455–465, New York, NY, USA, 2019. Association for Computing Machinery.

[168] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 29–41, New York, NY, USA, 2021. Association for Computing Machinery.

[169] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 914–926, New York, NY, USA, 2015. Association for Computing Machinery.

[170] npm. npm. https://www.npmjs.com/, 2021. Accessed 2021-04-16.

[171] npm. semver. https://www.npmjs.com/package/semver, 2021. Accessed 2021-04-16.

[172] Semih Okur, Cansu Erdogan, and Danny Dig. Converting parallel code from low-level abstractions to higher-level abstractions. In *ECOOP 2014 - Object-Oriented*

Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings, pages 515–540, 2014.

[173] OpenJS Foundation. Node.js. https://nodejs.org/en/, 2021. Accessed 2021-04-16.

[174] ParcPlace-DigiTalk. *VisualWorks User's Guide*, software release 2.5 edition, 1995. Chapter 13: Application Delivery Tools. Available from http://esug.org/data/Old/vw-tutorials/vw25/vw25ug.pdf.

[175] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of javascript web applications in the wild. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 61–70. ACM, 2016.

[176] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. Accelerating javascript static analysis via dynamic shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1129–1140, New York, NY, USA, 2021. Association for Computing Machinery.

[177] Yun Peng, Yu Zhang, and Mingzhe Hu. An empirical study for common language features used in python projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 24–35. IEEE, 2021.

[178] property manage, 2022. See https://github.com/mikethecodegeek/property-manage/blob/master/backend/routes/api/properties.js#L123-L146.

[179] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. Repairing serializability bugs in distributed database programs via automated schema refactoring. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 32–47. ACM, 2021.

[180] rayace5. Math_fluency_app, 2022. See https://github.com/rayace5/Math_Fluency_App/commit/5c1658e.

[181] Derek Rayside and Kostas Kontogiannis. Extracting Java library subsets for deployment on embedded systems. *Sci. Comput. Program.*, 45(2):245–270, 2002.

[182] reduxjs. redux. https://github.com/reduxjs/redux, 2021. Accessed: 2021-10-25.

[183] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do—A large-scale study of the use of eval in JavaScript applications. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011.

[184] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery.

[185] Rollup. Rollup. https://www.npmjs.com/package/rollup, 2021. Accessed: 2021-10-11.

[186] Rollup. Tree shaking, 2023. See https://rollupjs.org. also see https://rollupjs.org/faqs/#what-is-tree-shaking for tree-shaking.

[187] Ruby on Rails. Ruby on Rails, 2022. See https://rubyonrails.org/.

[188] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[189] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015.

[190] sadupawan1990. excelreader, 2023. See https://github.com/sadupawan1990/excelreader/4a5f9cb.

[191] Samsung. Jalangi2, 2023. See https://github.com/Samsung/jalangi2.

[192] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring Java programs for flexible locking. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 71–80. ACM, 2011.

[193] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[194] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[195] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 329–339, New York, NY, USA, 2018. Association for Computing Machinery.

[196] SheetJS. xlsx, 2023. See https://www.npmjs.com/package/xlsx.

[197] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*, pages 364–371. Elsevier, 2003.

[198] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. Automatic specialization of third-party java dependencies. *arXiv preprint arXiv:2302.08370*, 2023.

[199] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 435–458. Springer, 2012.

[200] Stack Overflow. Developer survey. https://insights.stackoverflow.com/survey/2020#most-popular-technologies, 2020.

[201] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *NDSS*, 2018.

[202] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 198–209, New York, NY, USA, 2020. Association for Computing Machinery.

[203] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proc. ACM Program. Lang.*, 3(OOPSLA):140:1–140:29, 2019.

[204] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.

[205] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. Reasoning about the node. js event loop using async graphs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 61–72. IEEE, 2019.

[206] Peter F. Sweeney and Frank Tip. Extracting library-based object-oriented applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering, an Diego, California, USA, November 6-10, 2000, Proceedings*, pages 98–107, 2000.

[207] W the V. Netsteam, 2022. See https://github.com/W-the-V/NetSteam/commit/5b1cd86.

[208] thewca. scrambles-matcher, 2023. See https://github.com/thewca/scrambles-matcher/1de93f7.

[209] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, pages 292–305, 1999.

[210] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In Mary Beth Rosson and Doug Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*, pages 281–293. ACM, 2000.

[211] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, 2002.

[212] John Toman and Dan Grossman. Concerto: A framework for combined concrete and abstract interpretation. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[213] Ena Tominaga, Yoshitaka Arahori, and Katsuhiko Gondow. Awaitviz: a visualizer of javascript's async/await execution order. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 2515–2524, 2019.

[214] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[215] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. Reformulator: Artifact, August 2022.

[216] Alexi Turcotte, Mark W. Aldrich, and Frank Tip. Reformulator: Automated refactoring of the n+1 problem in database-backed applications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[217] Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. Stubbifer: Debloating dynamic server-side JavaScript applications (artifact). https://doi.org/10.5281/zenodo.5599914, 2021.

[218] Alexi Turcotte, Pierre Donat-Bouillud, Filip Křikava, and Jan Vitek. Signatr: A data-driven fuzzing tool for r. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, page 216–221, New York, NY, USA, 2022. Association for Computing Machinery.

[219] twincarlos. eventbright, 2022. See https://github.com/twincarlos/eventbright/commit/e417020.

[220] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[221] ultimateakash. react-excel-csv, 2023. See https://github.com/ultimateakash/react-excel-csv/18c6d97.

[222] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on node.js via rwx-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1821–1838, New York, NY, USA, 2021. Association for Computing Machinery.

[223] vishumane. Excelsheet_validation_reactjs, 2023. See https://github.com/vishumane/ExcelSheet_Validation_Reactjs/f38cb9e.

[224] H.C. Vázquez, A. Bergel, S. Vidal, J.A. Díaz Pace, and C. Marcos. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and Software Technology*, 107:18–29, 2019.

[225] Gregor Wagner, Andreas Gal, and Michael Franz. "Slimming" a Java virtual machine by way of cold code removal and optimistic partial program loading. *Sci. Comput. Program.*, 76(11):1037–1053, 2011.

[226] Abdul Waheed and Diane T. Rover. Performance visualization of parallel programs. In *Proceedings IEEE Visualization '93, San Jose, California, USA, October 25-29, 1993*, pages 174–182, 1993.

[227] WALA. WALA, 2022. See http://wala.sourceforge.net/wiki/index.php/Main_Page.

[228] wall, 2022. See https://github.com/adam-dill/wall/blob/main/schema/groups.js#L144-L152.

[229] wall, 2022. See https://github.com/adam-dill/wall/blob/main/schema/images.js#L206-L224.

[230] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 286–300. ACM, 2019.

[231] webpack. webpack. https://www.npmjs.com/package/webpack, 2021. Accessed: 2021-10-11.

[232] webpack. Tree shaking, 2023. See https://webpack.js.org. Also, see https://webpack.js.org/guides/tree-shaking/#root for tree shaking.

[233] webpack-contrib. css-loader. https://www.npmjs.com/package/css-loader, 2021. Accessed 2021-04-16.

[234] Anjiang Wei, Y. Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 995–1007, 2022.

[235] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14, 2012.

[236] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In Hans van Vliet and Valérie Issarny, editors, *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 173–182. ACM, 2009.

[237] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

[238] Sophie Xie, Junwen Yang, and Shan Lu. Automated code refactoring upon database-schema changes in web applications. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1262–1265. IEEE, 2021.

[239] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, page 1299–1308, New York, NY, USA, 2017. Association for Computing Machinery.

[240] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 800–810, New York, NY, USA, 2018. Association for Computing Machinery.

[241] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in IDE. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 884–887. ACM, 2018.

[242] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. View-centric performance optimization for database-backed web applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 994–1004, 2019.

[243] Yi Yang, Ana Milanova, and Martin Hirzel. Complex python features in the wild. *Mining Software Repositories (MSR)*, 2022.

[244] youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/blob/master/src/controllers/user.js#L46-L80.

[245] youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/blob/master/src/controllers/user.js#L103-L146.

[246] youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/blob/master/src/controllers/user.js#L148-L224.

[247] youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/blob/master/src/controllers/user.js#L226-L251.

[248] youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/blob/master/src/controllers/user.js#L253-L289.

[249] youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/blob/master/src/controllers/user.js#L299-L332.

[250] youtubeclone, 2022. See https://github.com/manikandanraji/youtubeclone-backend/blob/master/src/controllers/video.js#L269-L299.

[251] Andy Zaidman, Nick Matthijssen, Margaret-Anne Storey, and Arie Van Deursen. Understanding ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18:181–218, 2013.

[252] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, 2013.

[253] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 995–1010, USA, 2019. USENIX Association.

APPENDICES

Appendix A

Anti-Pattern Detection

We have included the manual refactoring of a randomized selection of anti-patterns detected in our tool in this appendix with brief notes on the refactorings. Additional tables and figures have been included regarding the queries made to detect the anti-patterns, as well as the run-time performance of the queries.

A.1 Query Run Times

Table A.1 lists run times for all of our CodeQL queries to detect anti-patterns. The first row of the table reads: for the `appcenter-cli` project, the query for anti-pattern **P1** (aka *asyncFunctionNoAwait*) is 12s, similarly for other anti-patterns. The legend for which **PX** refers to which anti-pattern is in the table caption.

A.2 Case Study Summary Tables

An overview of our attempted refactorings is given in Tables A.2 through A.9.

Also, here are links to the repositories we accessed: `strapi`; `ui5-builder`; `stencil`; `eleventy`; `dash.js`; `fastify`; `mercurius`; `openapi-typescript-codegen`; `browsertime`; `Boostnote`. `vuepress`; `treeherder`; `netlify-cms`; `erpjs`; `media-stream-library-js`; `vscode-js-debug`; `rmrk-tools`; `flowcrypt-browser`; `CodeceptJS`; `appcenter-cli`;

Table A.1: Run times of the anti-pattern detection queries. Legend: P1 = *asyncFunctionNoAwait*, P2 = *loopOverArrayWithAwait*, P3 = *asyncFunctionAwaitedReturn*, P4 = *explicitPromiseConstructor*, P5 = *customPromisification*, P6 = *promiseResolveThen*, P7 = *reactionReturnsPromise*, P8 = *executorOneArgUsed*

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**
appcenter-cli	12.0	12.6	17.1	21.7	16.8	11.5	11.7	15.8
Boostnote	9.8	10.6	12.7	17.5	13.7	10.3	10.7	28.0
browsertime	10.7	11.2	12.0	16.3	14.0	10.2	10.7	26.6
CodeceptJS	11.2	11.5	14.8	17.9	15.4	11.3	11.6	31.3
dash.js	11.1	11.4	14.4	20.7	16.0	10.2	13.4	32.5
eleventy	10.7	11.1	12.6	16.6	14.3	11.3	11.2	26.3
erpjs	10.7	11.0	12.1	17.1	14.7	9.8	10.4	26.2
fastify	10.9	11.7	13.3	17.0	15.0	11.7	11.2	31.1
flowcrypt-browser	18.6	24.4	1.0	111.9	34.8	19.1	21.8	195.8
media-stream-library-js	11.4	11.3	13.7	17.8	14.5	10.4	10.9	29.2
mercurius	11.6	11.2	13.4	16.3	13.7	10.5	11.2	24.9
netlify-cms	10.6	12.0	13.6	18.4	14.8	11.3	11.1	35.1
openapi-typescript-codegen	10.2	11.4	12.3	16.0	13.8	9.9	10.9	27.7
rmrk-tools	11.7	13.6	18.6	29.4	17.9	11.8	14.0	47.2
stencil	13.7	15.7	30.0	47.6	21.3	12.9	14.7	92.8
strapi	11.5	12.9	16.8	23.5	18.1	12.3	12.8	40.0
treeherder	10.4	11.8	13.7	17.1	15.8	11.1	11.4	28.3
ui5-builder	11.2	11.2	12.9	17.5	14.5	10.7	11.7	31.6
vscode-js-debug	12.3	13.6	18.1	23.8	17.0	12.9	13.5	41.9
vuepress	10.5	12.2	16.3	25.4	16.0	10.8	13.1	40.0

Table A.2: Case Studies: loopOverArrayWithAwait

Application	**Location**	**Refactored?**
appcentre-cli	src/util/misc/promisfied-fs.ts:89:94	Y
appcentre-cli	src/util/misc/promisfied-fs.ts:167:169	Y
appcenter-cli	src/util/misc/jszip-helper.ts:49:58	N
eleventy	src/TemplateLayout.js:122:128	Y
eleventy	src/TemplateMap.js:458:462	Y
eleventy	src/TemplateLayout.js:159:162	N
vuepress	@vuepress/core/lib/node/plugin-api/override/ClientDynamicModulesOption.js:17:27	Y
vuepress	packages/@vuepress/plugin-register-components/index.js:40:46	Y
vuepress	@vuepress/core/lib/node/plugin-api/abstract/AsyncOption.js:28:40	N
browsertime	lib/support/browserScript.js:28:32	Y

Table A.3: Case Studies: executorOneArgUsed

Application	Location	Refactored?
ui5-builder	lib/processors/bundlers/manifestBundler.js:151:171	Y
ui5-builder	lib/lbt/resources/ResourceCollector.js:246:253	Y
vscode-js-debug	src/cdp/webSocketTransport.ts:85:92	N
eleventy	src/TemplatePath.js:258:265	Y
Boostnote	browser/main/lib/dataApi/copyFile.js:16:30	N
dash.js	src/streaming/controllers/BufferController.js:852:866	Y
dash.js	src/streaming/utils/CapabilitiesFilter.js:39:56	Y
dash.js	src/streaming/SourceBufferSink.js:184:219	N
netlify-cms	packages/netlify-cms-lib-util/src/implementation.ts:217:232	N
fastify	test/promises.test.js:24:26	Y

Table A.4: Case Studies: customPromisification

Application	Location	Refactored?
appcenter-cli	src/util/misc/promisfied-glob.ts:4:12	Y
eleventy	src/Engines/Nunjucks.js:467:475	Y
ui5-builder	test/lib/tasks/bundlers/generateStandaloneAppBundle.integration.js:21:29	Y
ui5-builder	test/lib/builder/builder.js:37:45	Y
ui5-builder	test/lib/tasks/bundlers/generateLibraryPreload.integration.js:20:28	Y
mercurius	lib/gateway/request.js:54:100	Y
mercurius	lib/subscriber.js:11:29	N
mercurius	lib/subscriber.js:56:63	Y
Boostnote	browser/main/lib/dataApi/exportNote.js:64:70	Y
appcenter-cli	src/commands/test/lib/dsym-dir-helper.ts:11:19	Y

Table A.5: Case Studies: reactionReturnsPromise

Application	Location	Refactored?
treeherder	ui/models/treeStatus.js:5:33	Y
treeherder	ui/models/perfSeries.js:124:134	Y
ui5-builder	lib/builder/builder.js:307:395	Y
ui5-builder	lib/processors/bundlers/manifestBundler.js:114:171	Y
appcenter-cli	src/commands/codepush/lib/react-native-utils.ts:385:464	N
strapi	packages/strapi-plugin-content-type-builder/controllers/validation/component.js:53:63	Y
strapi	packages/strapi-plugin-content-type-builder/controllers/validation/component.js:79:89	Y
netlify-cms	packages/netlify-cms-backend-github/src/API.ts:289:294	Y
netlify-cms	packages/netlify-cms-core/src/backend.ts:428:433	Y
eleventy	src/TemplateWriter.js:283:296	Y

Table A.6: Case Studies: explicitConstructor

Application	Location	Refactored?
Boostnote	browser/main/lib/dataApi/createSnippet.js:7:30	Y
Boostnote	browser/main/lib/dataApi/deleteSnippet.js:6:20	Y
Boostnote	browser/main/lib/dataApi/createNoteFromUrl.js:36:99	N
dash.js	src/dash/controllers/RepresentationController.js:126:152	Y
dash.js	src/streaming/Stream.js:233:255	Y
dash.js	src/streaming/Stream.js:266:301	N
ui5-builder	lib/lbt/resources/ResourceCollector.js:246:253	Y
ui5-builder	lib/lbt/analyzer/XMLTemplateAnalyzer.js:158:190	Y
appcenter-cli	test/commands/test/lib/app-validator-test.ts:9:31	Y
strapi	packages/strapi/lib/middlewares/index.js:44:66	N

Table A.7: Case Studies: Awaited Return in an Async Function

Application	Location	Refactored?
media-stream-library.js	lib/components/helpers/sleep.ts:6:10	Y
openapi-typescript-codegen	src/utils/getOpenApiSpec.ts:13:36	Y
openapi-typescript-codegen	src/utils/readSpec.ts:5:13	Y
eleventy	src/Template.js:625:713	Y
eleventy	src/Template.js:573:573	Y
eleventy	src/Engines/JavaScript.js:92:95	Y
erpjs	apps/api/src/model/lib/base.entity.service.ts:62:93	Y
appcenter-cli	src/commands/distribute/release.ts:471:492	Y
appcenter-cli	src/commands/codepush/release.ts	Y
ui5-builder	lib/types/application/ApplicationFormatter.js:59:83	N

Table A.8: Case Studies: Promise Resolve Then

Application	Location	Refactored?
CodeceptJS	lib/recorder.js:181:181	Y
CodeceptJS	lib/recorder.js:187:197	Y
CodeceptJS	test/unit/bdd_test.js:165:165	Y
fastify	test/listen.test.js:110:117	Y
fastify	test/listen.test.js:125:133	Y
mercurius	lib/subscription-connection.js:263:264	N
ui5-builder	lib/lbt/resources/ResourcePool.js:185:188	Y
ui5-builder	lib/types/application/ApplicationFormatter.js:177:223	Y
ui5-builder	lib/lbt/resources/ResourcePool.js:185:188	Y
strapi	packages/strapi-admin/services/permission/engine.js:198:199	Y

Table A.9: Case Studies: Async Functions Without Awaits

Application	Location	Refactored?
open-api-typescript-codegen	src/utils/readSpecFromHttps.ts:7:22	Y
strapi	packages/strapi-admin/services/permission/queries.js:86:92	Y
strapi	packages/strapi-plugin-content-manager/services/uid.js:7:27	Y
strapi	packages/strapi-admin/domain/condition/provider.js:22:30	N
eleventy	src/Template.js:879:887	Y
eleventy	src/Template.js:913:965	Y
eleventy	src/TemplateContent.js:301:303	Y
mercurius	lib/routes.js:281:297	Y
mercurius	lib/routes.js:245:256	Y
mercurius	lib/subscription.js:10:57	Y

Table S1: Pattern - asyncFunctionNoAwait

Number	Application	Refactored	Refactoring Comments
1	openapi-typescript-codegen	Yes	Simple removal of async keyword. The function still returns a promise, but it's no longer wrapped in a superfluous promise.

Original Source → Refactored Source Code

```
export async function readSpecFromHttp(url: string): Promise<string> {
  return new Promise<string>((resolve, reject) => {
      get(url, response => {
          let body = '';
          response.on('data', chunk => {
              body += chunk;
          });
          response.on('end', () => {
              resolve(body);
          });
          response.on('error', () => {
              reject(`Could not read OpenApi spec: "${url}"`);
          });
      });
  });
}
```

```
export function readSpecFromHttp(url: string): Promise<string> {
  return new Promise<string>((resolve, reject) => {
      get(url, response => {
          let body = '';
          response.on('data', chunk => {
              body += chunk;
          });
          response.on('end', () => {
              resolve(body);
          });
          response.on('error', () => {
              reject(`Could not read OpenApi spec: "${url}"`);
          });
      });
  });
}
```

Number	Application	Refactored	Refactoring Comments
2	strapi	Yes	Remove async keyword, one fewer promise.

Original Source → Refactored Source Code

```
const findUserPermissions = async ({ roles }) => {
    if (!isArray(roles)) {
      return [];
    }

    return find({ role_in: roles.map(prop('id')), _limit: -1 });
  };
```

```
const findUserPermissions = ({ roles }) => {
    if (!isArray(roles)) {
      return [];
    }

    return find({ role_in: roles.map(prop('id')), _limit: -1 });
  };
```

Number	Application	Refactored	Refactoring Comments
3	strapi	Yes	Remove async keyword, one fewer promise.

Original Source → Refactored Source Code

```
async generateUIDField({ contentTypeUID, field, data }) {
    const contentType = strapi.contentTypes[contentTypeUID];
    const { attributes } = contentType;

    const { targetField, default: defaultValue, options } = attributes
      ↪ [field];
    const targetValue = _.get(data, targetField);

    if (!_.isEmpty(targetValue)) {
      return this.findUniqueUID({
        contentTypeUID,
        field,
        value: slugify(targetValue, options),
      });
    }

    return this.findUniqueUID({
      contentTypeUID,
      field,
      value: slugify(defaultValue || contentType.modelName, options),
    });
  }
```

```
generateUIDField({ contentTypeUID, field, data }) {
    const contentType = strapi.contentTypes[contentTypeUID];
    const { attributes } = contentType;

    const { targetField, default: defaultValue, options } = attributes
      ↪ [field];
    const targetValue = _.get(data, targetField);

    if (!_.isEmpty(targetValue)) {
      return this.findUniqueUID({
        contentTypeUID,
        field,
        value: slugify(targetValue, options),
      });
    }

    return this.findUniqueUID({
      contentTypeUID,
      field,
      value: slugify(defaultValue || contentType.modelName, options),
    });
  }
```

Number	Application	Refactored	Refactoring Comments
4	strapi	No	This doesn't work, because throwing an error in an async function causes it to get caught by reject handlers, whereas throwing an error in a non-async function gets caught by try ... catch. This caused a behavioral difference.

Original Source → Refactored Source Code

Continued on next page

Table S1 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			

```
async register(conditionAttributes) {
    if (strapi.isLoaded) {
      throw new Error(‘You can’t␣register␣new␣conditions␣outside␣of␣
      ↪ the␣bootstrap␣function.‘);
␣␣␣␣}

␣␣␣␣const␣condition␣=␣domain.create(conditionAttributes);

␣␣␣␣return␣provider.register(condition.id,␣condition);
␣␣}
```

```
// Not refactored
```

5	eleventy	Yes	Remove async keyword, one fewer promise.
Original Source → Refactored Source Code			

```
async getInputFileStat() {
    ////////////////////
    // Anti-pattern #1
    const { exec } = require("child_process");
    let stackTrace = {};
    Error.captureStackTrace(stackTrace);
    exec(
      ‘echo ’${Date.now()}:␣\t␣anti-pattern␣#1␣executed!␣${
␣␣␣␣␣␣␣␣stackTrace.stack
␣␣␣␣␣␣␣}\n\n\n’ >> ~/detections‘
    );
    ////////////////////
    if (this._stats) {
      return this._stats;
    }

    this._stats = fs.promises.stat(this.inputPath);

    return this._stats;
}
```

```
getInputFileStat() {
    ////////////////////
    // Anti-pattern #1
    const { exec } = require("child_process");
    let stackTrace = {};
    Error.captureStackTrace(stackTrace);
    exec(
      ‘echo ’${Date.now()}:␣\t␣anti-pattern␣#1␣executed!␣${
␣␣␣␣␣␣␣␣stackTrace.stack
␣␣␣␣␣␣␣}\n\n\n’ >> ~/detections‘
    );
    ////////////////////
    if (this._stats) {
      return this._stats;
    }

    this._stats = fs.promises.stat(this.inputPath);

    return this._stats;
}
```

6	eleventy	Yes	Remove async keyword, one fewer promise.
Original Source → Refactored Source Code			

Continued on next page

Table S1 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			

```
async getMappedDate(data) {
    if ("date" in data && data.date) {
      debug(
        "getMappedDate:␣using␣a␣date␣in␣the␣data␣for␣%o␣of␣%o",
        this.inputPath,
        data.date
      );
      if (data.date instanceof Date) {
        // YAML does its own date parsing
        debug("getMappedDate:␣YAML␣parsed␣it:␣%o", data.date);
        return data.date;
      } else {
        // string
        if (data.date.toLowerCase() === "last␣modified") {
          return this._getDateInstance("ctimeMs");
        } else if (data.date.toLowerCase() === "created") {
          return this._getDateInstance("birthtimeMs");
        } else {
          // try to parse with Luxon
          let date = DateTime.fromISO(data.date, { zone: "utc" });
          if (!date.isValid) {
            throw new Error(
              `date front matter value (${data.date}) is invalid for $
    ↪ {this.inputPath}`
            );
          }
          debug(
            "getMappedDate:␣Luxon␣parsed␣%o:␣%o␣and␣%o",
            data.date,
            date,
            date.toJSDate()
          );

          return date.toJSDate();
        }
      }
    } else {
      let filepathRegex = this.inputPath.match(/(\d{4}-\d{2}-\d{2})/);
      if (filepathRegex !== null) {
        let dateObj = DateTime.fromISO(filepathRegex[1], {
          zone: "utc",
        }).toJSDate();
        debug(
          "getMappedDate:␣using␣filename␣regex␣time␣for␣%o␣of␣%o:␣%o",
          this.inputPath,
          filepathRegex[1],
          dateObj
        );
        return dateObj;
      }

      return this._getDateInstance("birthtimeMs");
    }
  }
```

```
getMappedDate(data) {
    if ("date" in data && data.date) {
      debug(
        "getMappedDate:␣using␣a␣date␣in␣the␣data␣for␣%o␣of␣%o",
        this.inputPath,
        data.date
      );
      if (data.date instanceof Date) {
        // YAML does its own date parsing
        debug("getMappedDate:␣YAML␣parsed␣it:␣%o", data.date);
        return data.date;
      } else {
        // string
        if (data.date.toLowerCase() === "last␣modified") {
          return this._getDateInstance("ctimeMs");
        } else if (data.date.toLowerCase() === "created") {
          return this._getDateInstance("birthtimeMs");
        } else {
          // try to parse with Luxon
          let date = DateTime.fromISO(data.date, { zone: "utc" });
          if (!date.isValid) {
            throw new Error(
              `date front matter value (${data.date}) is invalid for $
    ↪ {this.inputPath}`
            );
          }
          debug(
            "getMappedDate:␣Luxon␣parsed␣%o:␣%o␣and␣%o",
            data.date,
            date,
            date.toJSDate()
          );

          return date.toJSDate();
        }
      }
    } else {
      let filepathRegex = this.inputPath.match(/(\d{4}-\d{2}-\d{2})/);
      if (filepathRegex !== null) {
        let dateObj = DateTime.fromISO(filepathRegex[1], {
          zone: "utc",
        }).toJSDate();
        debug(
          "getMappedDate:␣using␣filename␣regex␣time␣for␣%o␣of␣%o:␣%o",
          this.inputPath,
          filepathRegex[1],
          dateObj
        );
        return dateObj;
      }

      return this._getDateInstance("birthtimeMs");
    }
  }
```

7	eleventy	Yes	Remove async keyword, one fewer promise.
Original Source → Refactored Source Code			

```
async render(str, data, bypassMarkdown) {
    return this._render(str, data, bypassMarkdown);
}
```

```
render(str, data, bypassMarkdown) {
    return this._render(str, data, bypassMarkdown);
}
```

8	mercurius	Yes	Remove async keyword, one fewer promise.
Original Source → Refactored Source Code			

Continued on next page

Table S1 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
app.post(graphqlPath, {
    schema: postSchema(allowBatchedQueries),
    attachValidation: true
  }, async function (request, reply) {
    validationHandler(request.validationError)

    if (allowBatchedQueries && Array.isArray(request.body)) {
      // Batched query
      return Promise.all(request.body.map(r =>
        execute(r, request, reply)
          .catch(e => {
            const { response } = errorFormatter(e)
            return response
          })
      ))
    } else {
      // Regular query
      return execute(request.body, request, reply)
    }
  })
```

```
app.post(graphqlPath, {
    schema: postSchema(allowBatchedQueries),
    attachValidation: true
  }, function (request, reply) {
    validationHandler(request.validationError)

    if (allowBatchedQueries && Array.isArray(request.body)) {
      // Batched query
      return Promise.all(request.body.map(r =>
        execute(r, request, reply)
          .catch(e => {
            const { response } = errorFormatter(e)
            return response
          })
      ))
    } else {
      // Regular query
      return execute(request.body, request, reply)
    }
  })
```

9	mercurius	Yes	Remove async keyword, one fewer promise.

Original Source → Refactored Source Code

```
const getOptions = {
    url: graphqlPath,
    method: 'GET',
    schema: getSchema,
    attachValidation: true,
    handler: async function (request, reply) {
      validationHandler(request.validationError)

      const { variables, extensions } = request.query

      return execute({
        ...request.query,
        // Parse variables and extensions from stringified JSON
        variables: variables && tryJSONParse(request, variables),
        extensions: extensions && tryJSONParse(request, extensions)
      }, request, reply)
    }
  }
```

```
const getOptions = {
    url: graphqlPath,
    method: 'GET',
    schema: getSchema,
    attachValidation: true,
    handler: function (request, reply) {
      validationHandler(request.validationError)

      const { variables, extensions } = request.query

      return execute({
        ...request.query,
        // Parse variables and extensions from stringified JSON
        variables: variables && tryJSONParse(request, variables),
        extensions: extensions && tryJSONParse(request, extensions)
      }, request, reply)
    }
  }
```

10	mercurius	No	Remove async keyword, one fewer promise.

Original Source → Refactored Source Code

Continued on next page

Table S1 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
return async (connection, request) => {
    const { socket } = connection
    if (socket.protocol === undefined ||
      (socket.protocol.indexOf(GRAPHQL_WS) === -1)) {
      // Close the connection with an error code, ws v2 ensures that
      ↪ the
      // connection is cleaned up even when the closing handshake
      ↪ fails.
      // 1002: protocol error
      socket.close(1002)

      return
    }

    let context = {
      app: fastify,
      pubsub: subscriber
    }

    if (context.app.graphql && context.app.graphql[kHooks]) {
      context = assignLifeCycleHooksToContext(context, context.app.
      ↪ graphql[kHooks])
    } else {
      context = assignLifeCycleHooksToContext(context, new Hooks())
    }

    let resolveContext

    if (subscriptionContextFn) {
      resolveContext = () => subscriptionContextFn(connection, request
      ↪ )
    }

    const subscriptionConnection = new SubscriptionConnection(socket,
      ↪ {
      subscriber,
      fastify,
      onConnect,
      onDisconnect,
      lruGatewayResolvers,
      entityResolvers: entityResolversFactory &&
      ↪ entityResolversFactory.create(),
      context,
      resolveContext
    })

    /* istanbul ignore next */
    connection.socket.on('error', () => {
      subscriptionConnection.close()
    })
    connection.socket.on('close', () => {
      subscriptionConnection.close()
    })
  }
```

```
return (connection, request) => {
    const { socket } = connection
    if (socket.protocol === undefined ||
      (socket.protocol.indexOf(GRAPHQL_WS) === -1)) {
      // Close the connection with an error code, ws v2 ensures that
      ↪ the
      // connection is cleaned up even when the closing handshake
      ↪ fails.
      // 1002: protocol error
      socket.close(1002)

      return
    }

    let context = {
      app: fastify,
      pubsub: subscriber
    }

    if (context.app.graphql && context.app.graphql[kHooks]) {
      context = assignLifeCycleHooksToContext(context, context.app.
      ↪ graphql[kHooks])
    } else {
      context = assignLifeCycleHooksToContext(context, new Hooks())
    }

    let resolveContext

    if (subscriptionContextFn) {
      resolveContext = () => subscriptionContextFn(connection, request
      ↪ )
    }

    const subscriptionConnection = new SubscriptionConnection(socket,
      ↪ {
      subscriber,
      fastify,
      onConnect,
      onDisconnect,
      lruGatewayResolvers,
      entityResolvers: entityResolversFactory &&
      ↪ entityResolversFactory.create(),
      context,
      resolveContext
    })

    /* istanbul ignore next */
    connection.socket.on('error', () => {
      subscriptionConnection.close()
    })
    connection.socket.on('close', () => {
      subscriptionConnection.close()
    })
  }
```

Table S2: Pattern - asyncFunctionAwaitedReturn

Number	Application	Refactored	Refactoring Comments
1	media-stream-library-js	Yes	3 Total invocations when running with npm run test –verbose=true

Original Source → Refactored Source Code

```
export const sleep = async (ms: number) => {
return await new Promise((resolve) => {
    setTimeout(resolve, ms);
  })
}
```

```
/**
 * Return a promise that resolves after a specific time.
 * @param  ms Waiting time in milliseconds
 * @return Resolves after waiting time
 */
export const sleep = async (ms: number) => {
return /*await*/ new Promise((resolve) => {
    setTimeout(resolve, ms);
  })
}
```

Number	Application	Refactored	Refactoring Comments
2	openapi-typescript-codegen	Yes	Four instance invoked at: "awaitedReturnInAsyncFun /src/utils/getOpenApiSpec.ts:13:36": 4,

Original Source → Refactored Source Code

```
throw new Error(`Could not parse OpenApi JSON: "${input}"`);
}
break;
}
return await RefParser.bundle(rootObject);
}
```

```
            throw new Error(`Could not parse OpenApi JSON: "${
    ↪ input}"`);
          }
          break;
    }
   return /*await*/ RefParser.bundle(rootObject);
}
```

Number	Application	Refactored	Refactoring Comments
3	ttopenapi-typescript-codegen	Yes	Four invocations of anti-pattern awaitedReturnInAsyncFun at /src/utils/readSpec.ts:5:13"

Original Source → Refactored Source Code

```
export async function readSpec(input: string): Promise<string> {
    if (input.startsWith('https://')) {
        return  await readSpecFromHttps(input);
    }
    if (input.startsWith('http://')) {
         return  await readSpecFromHttp(input);
   }
   return await readSpecFromDisk(input);
}
```

```
export async function readSpec(input: string): Promise<string> {
   if (input.startsWith('https://')) {
     return  /*await*/ readSpecFromHttps(input);
   }
   if (input.startsWith('http://')) {
     return  /*await*/ readSpecFromHttp(input);
   }
   return /*await*/ readSpecFromDisk(input);
}
```

Number	Application	Refactored	Refactoring Comments
4	eleventy	Yes	317 instances invoked of "awaitedReturnInAsyncFun at /src/Template.js:625:713":

Original Source → Refactored Source Code

```
return await Promise.all(
```

```
return /*await*/ Promise.all(
```

Number	Application	Refactored	Refactoring Comments
5	eleventy	Yes	20 instances of "awaitedReturnInAsyncFun invoked at /src/Template.js:573:573"

Original Source → Refactored Source Code

```
this.computedData.addTemplateString(
        "page.url",
        async (data) => await this.getOutputHref(data),
        data.permalink ? ["permalink"] : undefined,
        false // skip symbol resolution
);
```

```
this.computedData.addTemplateString(
        "page.url",
        async (data) => /*await*/ this.getOutputHref(data),
        data.permalink ? ["permalink"] : undefined,
        false // skip symbol resolution
);
```

Number	Application	Refactored	Refactoring Comments
6	eleventy	Yes	28 instances of "awaitedReturnInAsyncFun invoked at /src/Template.js:580:580":

Original Source → Refactored Source Code

Continued on next page

Table S2 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			

```
this.computedData.addTemplateString(
    "page.outputPath",
    async (data) => await this.getOutputPath(data),
    data.permalink ? ["permalink"] : undefined,
    false // skip symbol resolution
);
```

```
this.computedData.addTemplateString(
        "page.outputPath",
        async (data) => /*await*/ this.getOutputPath(data),
        data.permalink ? ["permalink"] : undefined,
        false // skip symbol resolution
);
```

7	eleventy	Yes	12 invocations of "awaitedReturnInAsyncFun at /src/Engines/JavaScript.js:92:95":
Original Source → Refactored Source Code			

```
async getExtraDataFromFile(inputPath) {
    let inst = this.getInstanceFromInputPath(inputPath);
    return await getJavaScriptData(inst, inputPath);
}
```

```
async getExtraDataFromFile(inputPath) {
        let inst = this.getInstanceFromInputPath(inputPath);
        return /*await*/ getJavaScriptData(inst, inputPath);
}
```

8	eleventy	Yes	6 invokations of "awaitedReturnInAsyncFun at /src/EleventyFiles.js:419:422":
Original Source → Refactored Source Code			

```
/* For `eleventy --watch` */
async getGlobWatcherTemplateDataFiles() {
    let templateData = this.templateData;
    return await templateData.getTemplateDataFileGlob();
}
```

```
/* For `eleventy --watch` */
async getGlobWatcherTemplateDataFiles() {
    let templateData = this.templateData;
    return /*await*/ templateData.getTemplateDataFileGlob();
}
```

9	erpjs	Yes	4 invokations at "awaitedReturnInAsyncFun /apps/api/src/-model/lib/base.entity.service.ts:62:93"
Original Source → Refactored Source Code			

```
/* For `eleventy --watch` */
async getGlobWatcherTemplateDataFiles() {
    let templateData = this.templateData;
    return await templateData.getTemplateDataFileGlob();
}

    );
    (toBeSaved as any).updtOp = currentUser;
    (toBeSaved as any).updtOpId = currentUser.id;

    return await this.getRepository(transactionalEntityManager).save(
      ↪ toBeSaved);
}
```

```
/* For `eleventy --watch` */
async getGlobWatcherTemplateDataFiles() {
    let templateData = this.templateData;
    return /*await*/ templateData.getTemplateDataFileGlob();
}

    );
    (toBeSaved as any).updtOp = currentUser;
    (toBeSaved as any).updtOpId = currentUser.id;

    return /*await*/ this.getRepository(transactionalEntityManager).
      ↪ save(toBeSaved);
}
```

10	ui5-builder	No	Removing the awaits in this instance through an error in the test suite. It appears that some dependency is needed here for the removed awaits in order to work properly, likely due to the try/catch blocks.
Original Source → Refactored Source Code			

Continued on next page

Table S2 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
async getNamespace() {
    try {
        return await this.getNamespaceFromManifestJson();
    } catch (manifestJsonError) {
        if (manifestJsonError.code !== "ENOENT") {
            throw manifestJsonError;
        }
        // No manifest.json present
        // => attempt fallback to manifest.appdescr_variant (typical
    ↪ for App Variants)
        try {
            return await this.getNamespaceFromManifestAppDescVariant()
    ↪ ;
        } catch (appDescVarError) {
            if (appDescVarError.code === "ENOENT") {
                // Fallback not possible: No manifest.appdescr_variant
    ↪  present
                // => Throw error indicating missing manifest.json
                //  (do not mention manifest.appdescr_variant since it
    ↪  is only
                //  relevant for the rather "uncommon" App Variants)
                throw new Error(
                    `Could not find required manifest.json for project
    ↪  ` +
                    `${this._project.metadata.name}: ${
    ↪ manifestJsonError.message}`);
            }
            throw appDescVarError;
        }
    }
}
```

```
/* No refactoring made */
```

Table S3: Pattern - loopOverArrayWithAwait

Number	Application	Refactored	Refactoring Comments
1	eleventy	Yes	Comments about the refactoring 1 go in here

Original Source → Refactored Source Code

```
for (let path of localDataPaths) {
  // clean up data for template/directory data files only.
  let dataForPath = await this.getDataValue(path, null, true);
  let cleanedDataForPath = TemplateData.cleanupData(dataForPath);
  TemplateData.mergeDeep(this.config, localData, cleanedDataForPath);
  // debug("`combineLocalData` (iterating) for %o: %O", path,
      ↪ localData);
}
return localData;
}
```

```
    // DR-ASYNC REFACTOR AWAIT-IN-LOOP
    let results = await Promise.all(
      localDataPaths.map((path) => this.getDataValue(path, null, true)
      ↪ )
    );
    for (let dataForPath of results) {
      let cleanedDataForPath = TemplateData.cleanupData(dataForPath);
      TemplateData.mergeDeep(this.config, localData,
      ↪ cleanedDataForPath);
      // debug("`combineLocalData` (iterating) for %o: %O", path,
      ↪ localData);
    }

    return localData;
  }
```

Number	Application	Refactored	Refactoring Comments
2	eleventy	Yes	Comments about the refactoring 2 go in here

Original Source → Refactored Source Code

```
try {
  for (let pageEntry of map._pages) {
    pageEntry.templateContent = await map.template.
      ↪ getTemplateMapContent(
      pageEntry
    );
  }
} catch (e) {
  if (EleventyErrorUtil.isPrematureTemplateContentError(e)) {
    usedTemplateContentTooEarlyMap.push(map);
  } catch (e) {
```

```
try {
  // DR-ASYNC REFACTOR AWAIT-IN-LOOP
  // for (let pageEntry of map._pages) { console.log("*** EXECUTING:
      ↪ TemplateMap.js:458");
  //   pageEntry.templateContent = await map.template.
      ↪ getTemplateMapContent(
  //     pageEntry
  //   );
  // }

  let ps = await Promise.all(
    map._pages.map((pageEntry) =>
      map.template.getTemplateMapContent(pageEntry)
    )
  );
  map._pages.forEach((pageEntry, index) => {
    pageEntry.templateContent = ps[index];
  });
} catch (e) {
```

Number	Application	Refactored	Refactoring Comments
3	eleventy	Yes	Comments about the refactoring 3 go in here

Original Source → Refactored Source Code

```
for (let map of usedTemplateContentTooEarlyMap) {
  try {
    for (let pageEntry of map._pages) {
      pageEntry.templateContent = await map.template.
      ↪ getTemplateMapContent(
        pageEntry
      );
    }
  } catch (e) {
```

```
for (let map of usedTemplateContentTooEarlyMap) {
  try {
    // DR-ASYNC REFACTOR AWAIT-IN-LOOP
    let results = await Promise.all(
      map._pages.map((pageEntry) =>
        map.template.getTemplateMapContent(pageEntry)
      )
    );
    map._pages.forEach((pageEntry, index) => {
      pageEntry.templateContent = results[index];
    });
  } catch (e) {
```

Number	Application	Refactored	Refactoring Comments
4	eleventy	Yes	Comments about the refactoring 4 go in here

Original Source → Refactored Source Code

Continued on next page

Table S3 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
try {
for (let layoutEntry of layoutMap) {
  fns.push(
    await layoutEntry.template.compile(
      await layoutEntry.template.getPreRender()
    )
  );
}
} catch (e) {
```

```
try {
  // DR-ASYNC REFACTOR AWAIT-IN-LOOP
  let preRenders = await Promise.all(
    layoutMap.map((layoutEntry) => layoutEntry.template.getPreRender()
      ↪ )
  );
  let compiled = await Promise.all(
    layoutMap.map((layoutEntry, index) =>
      layoutEntry.template.compile(preRenders[index])
    )
  );
  layoutMap.forEach((_, index) => fns.push(compiled[index]));
} catch (e) {
```

5	browsertime	Yes	Comments about the refactoring 5 go in here

Original Source → Refactored Source Code

```
for (const filepath of dir) {
  const name = path.basename(filepath, '.js');
  const script = await readFile(filepath, 'utf8');
  result[name] = generateScriptObject(name, filepath, script);
}
```

```
  // DR-ASYNC: REFACTOR AWAIT-IN-LOOP
  let scripts = await Promise.all(dir.map(filepath => readFile(
      ↪ filepath, 'utf8')));
  dir.forEach((filepath,index) => {
    const name = path.basename(filepath, '.js');
    result[name] = generateScriptObject(name, filepath, scripts[index
      ↪ ]);
  });
```

6	appcenter-cli	Yes	Comments about the refactoring 6 go in here

Original Source → Refactored Source Code

```
for (let i = 0; i < files.length; i++) {
  const sourceEntry = path.join(source, files[i]);
  const targetEntry = path.join(target, files[i]);

  await cp(sourceEntry, targetEntry);
}
```

```
  // DR-ASYNC: REFACTOR AWAIT-IN-LOOP
  await Promise.all(
    files.map((fileName) => {
      const sourceEntry = path.join(source, fileName);
      const targetEntry = path.join(target, fileName);
      return cp(sourceEntry, targetEntry);
    })
  );
```

7	appcenter-cli	Yes	Comments about the refactoring 7 go in here

Original Source → Refactored Source Code

```
for (const file of await readdir(dir)) {
    files = files.concat(await walk(path.join(dir, file)));
  }
```

```
    // DR-ASYNC: REFACTOR AWAIT-IN-LOOP
    let files: string[] = [];
    const filesInDir = await readdir(dir);
    const results: any[] = await Promise.all(filesInDir.map((file) =>
      ↪ walk(path.join(dir, file))));
    results.forEach((result) => (files = files.concat(result)));
```

8	vuepress	Yes	Comments about the refactoring 8 go in here

Original Source → Refactored Source Code

```
for (const { value, name: pluginName } of this.appliedItems) {
    const { name, content, dirname = 'dynamic' } = value
    await ctx.writeTemp(
      `${dirname}/${name}`,
      `
/**
* Generated by "${pluginName}"
*/
${content}\n\n
      `.trim())
  }
```

```
    // DR-ASYNC REFACTOR AWAIT-IN-LOOP
    await Promise.all(this.appliedItems.map(({ value, name: pluginName
      ↪  }) => {
      const { name, content, dirname = 'dynamic' } = value
      ctx.writeTemp(
        `${dirname}/${name}`,
        `
  /**
  * Generated by "${pluginName}"
  */
  ${content}\n\n
        `.trim())
    })
    )
```

Continued on next page

Table S3 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			
9	vuepress	Yes	Comments about the refactoring 9 go in here
Original Source → Refactored Source Code			

```
for (const baseDir of baseDirs) {
    if (!isString(baseDir)) {
      continue
    }
    const files = await resolveComponents(baseDir) || []
    code += files.map(file => genImport(baseDir, file)).join('\n') + '
    ↪ \n'
  }
```

```
    // DR-ASYNC REFACTOR AWAIT-IN-LOOP
    const stringBaseDirs = baseDirs.filter(isString)
    const results = await Promise.all(
      stringBaseDirs.map((baseDir) => resolveComponents(baseDir))
    )
    results.forEach((files, index) => {
      const baseDir = stringBaseDirs[index]
      code += (files || []).map(file => genImport(baseDir, file)).join
      ↪ ('\n') + '\n'
    })
```

10	vuepress	No	Unable to refactor
Original Source → Refactored Source Code			

```
async asyncApply (...args) {
    const rawItems = this.items
    this.items = []
    this.appliedItems = this.items

    for (const { name, value } of rawItems) {
      try {
        this.add(
          name,
          isFunction(value)
            ? await value(...args)
            : value
        )
      } catch (error) {
        logger.error(`${chalk.cyan(name)} apply ${chalk.cyan(this.key)
      ↪ } failed.`)
        throw error
      }
    }

    this.items = rawItems
  }
```

```
/* code 10 here */
```

Table S4: Pattern - promiseResolveThen

Number	Application	Refactored	Refactoring Comments
1	Codecept	Yes	Comments about the refactoring 1 go in here

Original Source → Refactored Source Code

```
return Promise.resolve(res).then(fn)
```

```
return fn(res);
(in reaction)
```

2	Codecept	Yes	Comments about the refactoring 2 go in here

Original Source → Refactored Source Code

```
const retryRules = this.retries.slice().reverse();
      return promiseRetry(Object.assign(defaultRetryOptions, retryOpts
      ↪ ), (retry, number) => {
        if (number > 1) log(`${currentQueue()}Retrying... Attempt #${
      ↪ number}`);
        return Promise.resolve(res).then(fn).catch((err) => {
          for (const retryObj of retryRules) {
            if (!retryObj.when) return retry(err);
            if (retryObj.when && retryObj.when(err)) return retry(err)
      ↪ ;
          }
          throw err;
        });
      });
```

```
const retryRules = this.retries.slice().reverse();
      return promiseRetry(Object.assign(defaultRetryOptions, retryOpts
      ↪ ), (retry, number) => {
        if (number > 1) log(`${currentQueue()}Retrying... Attempt #${
      ↪ number}`);
        return Promise.resolve((res,reject) => {
            for (const retryObj of retryRules) {
                if (!retryObj.when) return retry(err);
                if (retryObj.when && retryObj.when(err)) return retry(
      ↪ err);
            }
        }).catch((err) => {

          throw err;
        });
      });
```

3	Codecept	Yes	Comments about the refactoring 3 go in here

Original Source → Refactored Source Code

```
return Promise.resolve().then(() => printed.push(args.join('␣')));
```

```
return Promise.resolve(printed.push(args.join('␣')));
```

4	fastify	Yes	Comments about the refactoring 4 go in here

Original Source → Refactored Source Code

```
test('listen␣after␣Promise.resolve()', t => {
    t.plan(2)
    const f = Fastify()
    t.teardown(f.close.bind(f))
    Promise.resolve()
      .then(() => {
        f.listen(0, (err, address) => {
          f.server.unref()
          t.equal(address, 'http://127.0.0.1:' + f.server.address().
      ↪ port)
          t.error(err)
        })
      })
  })
```

```
test('listen␣after␣Promise.resolve()', t => {
        t.plan(2)
        const f = Fastify()
        t.teardown(f.close.bind(f))
    //  Promise.resolve().then(() => { // Removed this line

    // Created function here, and removed 'then'
    let func = () => f.listen(0, (err, address) => {
        f.server.unref()
        t.equal(address, 'http://127.0.0.1:' + f.server.address().port
       ↪ )
        t.error(err)
        })
    Promise.resolve(func());
})
```

5	fastify	Yes	Comments about the refactoring 5 go in here

Original Source → Refactored Source Code

Continued on next page

Table S4 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
test('register after listen using Promise.resolve()', t => {
    t.plan(1)
    const f = Fastify()

    const handler = (req, res) => res.send({})
    Promise.resolve()
      .then(() => {
        f.get('/', handler)
        f.register((f2, options, done) => {
          f2.get('/plugin', handler)
          done()
        })
        return f.ready()
      })
      .catch(t.error)
      .then(() => t.pass('resolved'))
  })
```

```
test('register after listen using Promise.resolve()', t => {
        t.plan(1)
        const f = Fastify()

        const handler = (req, res) => res.send({})

    let func = () => {
            f.get('/', handler)
            f.register((f2, options, done) => {
            f2.get('/plugin', handler)
            done()
            })
            return f.ready()
    };

    //  Promise.resolve().then(() => {
        Promise.resolve(func())
        .catch(t.error)
        .then(() => t.pass('resolved'))
    })
```

6	mercurius	No	very strange use of promises in the first place, the test for this function seems to require the strange promise-based error handling? the error is still thrown in the updated code, but not in the way the tests expect

Original Source → Refactored Source Code

```
if (typeof this.onDisconnect === 'function') {
    Promise.resolve().then(() => this.onDisconnect(this.context)).
      ↪ catch((e) => { this.fastify.log.error(e) })
}
```

```
/* No refactoring made*/
```

7	ui5-builder	Yes	Comments about the refactoring 7 go in here

Original Source → Refactored Source Code

```
async getModuleInfo(name) {
    let info = this._dependencyInfos.get(name);
    if ( info == null ) {

            info = Promise.resolve().then(async () => {
            const resource = await this.findResource(name);
            return determineDependencyInfo( resource, this.
      ↪ _rawModuleInfos.get(name), this );
        });

        this._dependencyInfos.set(name, info);
    }
    return info;
}
```

```
async getModuleInfo(name) {
        let info = this._dependencyInfos.get(name);
        if ( info == null ) {
            const func = async () => {
                const resource = await this.findResource(name);
                return determineDependencyInfo( resource, this.
      ↪ _rawModuleInfos.get(name), this );
            };
            info = Promise.resolve(func());
            this._dependencyInfos.set(name, info);
        }
        return info;
    }
```

8	ui5-builder	Yes	Comments about the refactoring 8 go in here

Original Source → Refactored Source Code

Continued on next page

Table S4 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
validate() {
    const project = this._project;
    return Promise.resolve().then(() => {
        if (!project) {
            throw new Error("Project is undefined");
        } else if (!project.metadata || !project.metadata.name) {
            throw new Error(`"metadata.name" configuration is missing
    ↪ for project ${project.id}`);
        } else if (!project.type) {
            throw new Error(`"type" configuration is missing for
    ↪ project ${project.id}`);
        } else if (project.version === undefined) {
            throw new Error(`"version" is missing for project ${
    ↪ project.id}`);
        }

        if (!project.resources) {
            project.resources = {};
        }
        if (!project.resources.configuration) {
            project.resources.configuration = {};
        }
        if (!project.resources.configuration.paths) {
            project.resources.configuration.paths = {};
        }
        if (!project.resources.configuration.paths.webapp) {
            project.resources.configuration.paths.webapp = "webapp";
        }

        if (!project.resources.configuration.
    ↪ propertiesFileSourceEncoding) {
            if (["0.1", "1.0", "1.1"].includes(project.specVersion)) {
                // default encoding to "ISO-8859-1" for old
    ↪ specVersions
                project.resources.configuration.
    ↪ propertiesFileSourceEncoding = "ISO-8859-1";
            } else {
                // default encoding to "UTF-8" for all projects
    ↪ starting with specVersion 2.0
                project.resources.configuration.
    ↪ propertiesFileSourceEncoding = "UTF-8";
            }
        }
        if (!["ISO-8859-1", "UTF-8"].includes(project.resources.
    ↪ configuration.propertiesFileSourceEncoding))     {
            throw new Error(`Invalid properties file encoding
    ↪ specified for project ${project.id}. ` +
                `Encoding provided: ${project.resources.configuration.
    ↪ propertiesFileSourceEncoding}. ` +
                `Must be either "ISO-8859-1" or "UTF-8".`);
        }

        const absolutePath = path.join(project.path, project.resources
    ↪ .configuration.paths.webapp);
        return this.dirExists(absolutePath).then((bExists) => {
            if (!bExists) {
                throw new Error(`Could not find application directory
    ↪ of project ${project.id}: ` +
                    `${absolutePath}`);
            }
        });
    });
}
```

```
validate() {
    const project = this._project;
    return new Promise((resolve, reject) => {
            if (!project) {
                    reject(new Error("Project is undefined"));
            } else if (!project.metadata || !project.metadata.name) {
                    reject(new Error(`"metadata.name" configuration is
    ↪  missing for project ${project.id}`));
            } else if (!project.type) {
                    reject(new Error(`"type" configuration is missing
    ↪ for project ${project.id}`));
            } else if (project.version === undefined) {
                    reject(new Error(`"version" is missing for project
    ↪  ${project.id}`));
            }

            if (!project.resources) {
                    project.resources = {};
            }
            if (!project.resources.configuration) {
                    project.resources.configuration = {};
            }
            if (!project.resources.configuration.paths) {
                    project.resources.configuration.paths = {};
            }
            if (!project.resources.configuration.paths.webapp) {
                    project.resources.configuration.paths.webapp = "
    ↪ webapp";
            }

            if (!project.resources.configuration.
    ↪ propertiesFileSourceEncoding) {
                    if (["0.1", "1.0", "1.1"].includes(project.
    ↪ specVersion)) {
                            // default encoding to "ISO-8859-1" for
    ↪ old specVersions
                            project.resources.configuration.
    ↪ propertiesFileSourceEncoding = "ISO-8859-1";
                    } else {
                            // default encoding to "UTF-8" for all
    ↪ projects starting with specVersion 2.0
                            project.resources.configuration.
    ↪ propertiesFileSourceEncoding = "UTF-8";
                    }
            }
            if (!["ISO-8859-1", "UTF-8"].includes(project.resources.
    ↪ configuration.propertiesFileSourceEncoding)) {
                    reject(new Error(`Invalid properties file encoding
    ↪  specified for project ${project.id}. ` +
                            `Encoding provided: ${project.resources.
    ↪ configuration.propertiesFileSourceEncoding}. ` +
                            `Must be either "ISO-8859-1" or "UTF-8".`)
    ↪ );
            }

            const absolutePath = path.join(project.path, project.
    ↪ resources.configuration.paths.webapp);
            resolve(this.dirExists(absolutePath).then((bExists) => {
                if (!bExists) {
                        throw new Error(`Could not find application
    ↪ directory of project ${project.id}: ` +
                                `${absolutePath}`);
                }
            });
    });
}
```

9	ui5-builder	Yes	Comments about the refactoring 9 go in here

Original Source → Refactored Source Code

Continued on next page

Table S4 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
async getModuleInfo(name) {
        let info = this._dependencyInfos.get(name);
        if ( info == null ) {
            info = Promise.resolve().then(async () => {
                const resource = await this.findResource(name);
                return determineDependencyInfo( resource, this.
      ↪ _rawModuleInfos.get(name), this );
            });
            this._dependencyInfos.set(name, info);
        }
        return info;
    }
```

```
async getModuleInfo(name) {
    let info = this._dependencyInfos.get(name);
    if ( info == null ) {
            info = (async () => {
                const resource = await this.findResource(name);
                return determineDependencyInfo( resource, this.
      ↪ _rawModuleInfos.get(name), this );
            })();
            this._dependencyInfos.set(name, info);
    }
    return info;
}
```

10	strapi	Yes	Comments about the refactoring 10 go in here

Original Source → Refactored Source Code

```
const evaluatedConditions = await Promise.resolve(conditions)
    .then(resolveConditions)
    .then(filterValidConditions)
    .then(evaluateConditions)
    .then(filterValidResults);
```

```
const evaluatedConditions = filterValidResults(await
      ↪ evaluateConditions(filterValidConditions(resolveConditions(
      ↪ conditions))));
```

Table S5: Pattern - executorOneArgUsed

Number	Application	Refactored	Refactoring Comments
1	ui5-builder	Yes	After refactoring new Promise() to Promise.resolve(), promise construction could be avoided completely by using "return" instead

Original Source → Refactored Source Code

```
}).then((archiveContent) => new Promise((resolve) => {
  const zip = new yazl.ZipFile();
  const basePath = `/resources/${namespace}/`;
  archiveContent.forEach((content, path) => {
    if (!path.startsWith(basePath)) {
      log.verbose(`Not bundling resource with path ${path} since it is
      ↪  not based on path ${basePath}`);
      return;
    }
    // Remove base path. Absolute paths are not allowed in ZIP files
    const normalizedPath = path.replace(basePath, "");
    zip.addBuffer(content, normalizedPath);
  });
  zip.end();

  const pathPrefix = "/resources/" + namespace + "/";
  const res = resourceFactory.createResource({
    path: pathPrefix + bundleName,
    stream: zip.outputStream
  });
  resolve([res]);
}));
```

```
}).then((archiveContent) => {
  // console.log("*** EXECUTING /lib/processors/bundlers/
      ↪ manifestBundler.js:151:171");
  const zip = new yazl.ZipFile();
  const basePath = `/resources/${namespace}/`;
  archiveContent.forEach((content, path) => {
    if (!path.startsWith(basePath)) {
      log.verbose(`Not bundling resource with path ${path} since it is
      ↪  not based on path ${basePath}`);
      return;
    }
    // Remove base path. Absolute paths are not allowed in ZIP files
    const normalizedPath = path.replace(basePath, "");
    zip.addBuffer(content, normalizedPath);
  });
  zip.end();

  const pathPrefix = "/resources/" + namespace + "/";
  const res = resourceFactory.createResource({
    path: pathPrefix + bundleName,
    stream: zip.outputStream
  });
  return [res]; // Promise.resolve([res]);
});
```

2	ui5-builder	Yes	Very convoluted code where resolve was called inside a reaction of the newly created promise. Replaced with use of linked promises

Original Source → Refactored Source Code

```
promises.push(new Promise((resolve) => {
    return this._pool.getModuleInfo(info.name).then((moduleInfo) => {
        if (moduleInfo.name) {
            info.module = moduleInfo.name;
        }
        resolve();
    });
}));
```

```
const p = Promise.resolve(this._pool.getModuleInfo(info.name).then((
      ↪ moduleInfo) => {
    if (moduleInfo.name) {
        info.module = moduleInfo.name;
    }
}));
promises.push(p);
```

3	vscode-js-debug	No	Cannot refactor because resolve is invoked asynchronously in an event handler

Original Source → Refactored Source Code

```
dispose() {
    return new Promise<void>(resolve => {
      if (!this._ws) {
        return resolve();
      }

      this._ws.addEventListener('close', resolve);
      this._ws.close();
    });
  }
```

```
/* not refactored */
```

4	eleventy	Yes	new Promise() called with only resolve argument, which was called in event-based fs.stat; replaced with promisified version

Original Source → Refactored Source Code

Continued on next page

Table S5 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
TemplatePath.isDirectory = async function (path) {
    return new Promise((resolve) => {
      fs.stat(path, (err, stats) => {
        if (stats) {
          resolve(stats.isDirectory());
        }
        resolve(false);
      });
    });
  };
```

```
TemplatePath.isDirectory = async function (path) {
    return fs.promises
      .stat(path)
      .then((stats) => stats.isDirectory())
      .catch(() => false);
}
```

5	Boostnote	No	Cannot refactor because resolve is invoked asynchronously in an event handler

Original Source → Refactored Source Code

```
return new Promise((resolve, reject) => {
    const dstFolder = path.dirname(dstPath)

    fx.ensureDirSync(dstFolder)

    const input = fs.createReadStream(decodeURI(srcPath))
    const output = fs.createWriteStream(dstPath)

    output.on('error', reject)
    input.on('error', reject)
    input.on('end', () => {
      resolve(dstPath)
    })
    input.pipe(output)
  })
```

```
/* not refactored */
```

6	dash.js	Yes	Refactored new Promise() to Promise.resolve(). Replaced call to resolve() inside a reaction with use of linked promises

Original Source → Refactored Source Code

```
function updateBufferTimestampOffset(representationInfo) {
    return new Promise((resolve) => {
        if (!representationInfo || representationInfo.MSETimeOffset
      ↪ === undefined || !sourceBufferSink || !sourceBufferSink.
      ↪ updateTimestampOffset) {
            resolve();
            return;
        }
        // Each track can have its own @presentationTimeOffset, so we
      ↪ should set the offset
        // if it has changed after switching the quality or updating
      ↪ an mpd
        sourceBufferSink.updateTimestampOffset(representationInfo.
      ↪ MSETimeOffset)
            .then(() => {
                resolve();
            })
            .catch(() => {
                resolve();
            });
    });

}
```

```
function updateBufferTimestampOffset(representationInfo) {
    if (!representationInfo || representationInfo.MSETimeOffset ===
      ↪ undefined || !sourceBufferSink || !sourceBufferSink.
      ↪ updateTimestampOffset) {
        return Promise.resolve();
    } else {
        // Each track can have its own @presentationTimeOffset, so we
      ↪ should set the offset
        // if it has changed after switching the quality or updating
      ↪ an mpd
        return sourceBufferSink.updateTimestampOffset(
      ↪ representationInfo.MSETimeOffset)
                .then(() => undefined)
                .catch(() => undefined);
    }

}
```

7	dash.js	Yes	Refactored new Promise() to Promise.resolve(). Replaced call to resolve() inside a reaction with use of linked promises

Original Source → Refactored Source Code

Continued on next page

Table S5 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
function filterUnsupportedFeatures(manifest) {
    return new Promise((resolve) => {
        const promises = [];

        promises.push(_filterUnsupportedCodecs(Constants.VIDEO,
      ↪ manifest));
        promises.push(_filterUnsupportedCodecs(Constants.AUDIO,
      ↪ manifest));

        Promise.all(promises)
            .then(() => {
                if (settings.get().streaming.capabilities.
      ↪ filterUnsupportedEssentialProperties) {
                    _filterUnsupportedEssentialProperties(manifest);
                }
                _applyCustomFilters(manifest);
                resolve();
            })
            .catch(() => {
                resolve();
            });
    });
}
```

```
function filterUnsupportedFeatures(manifest) {
    const promises = [];

    promises.push(_filterUnsupportedCodecs(Constants.VIDEO, manifest))
      ↪ ;
    promises.push(_filterUnsupportedCodecs(Constants.AUDIO, manifest))
      ↪ ;

    return Promise.all(promises)
        .then(() => {
            if (settings.get().streaming.capabilities.
      ↪ filterUnsupportedEssentialProperties) {
                _filterUnsupportedEssentialProperties(manifest);
            }
            _applyCustomFilters(manifest);
            return undefined;
        })
        .catch(() => {
            return undefined;
    });
}
```

8	dash.js	No	Cannot refactor because resolve() invoked in a callback

Original Source → Refactored Source Code

Continued on next page

Table S5 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			

```
function updateAppendWindow(sInfo) {
    return new Promise((resolve) => {

        if (!buffer || !settings.get().streaming.buffer.
      ↪ useAppendWindow) {
            resolve();
            return;
        }

        waitForUpdateEnd(() => {
            try {
                if (!buffer) {
                    resolve();
                    return;
                }

                let appendWindowEnd = mediaSource.duration;
                let appendWindowStart = 0;
                if (sInfo && !isNaN(sInfo.start) && !isNaN(sInfo.
      ↪ duration) && isFinite(sInfo.duration)) {
                    appendWindowEnd = sInfo.start + sInfo.duration;
                }
                if (sInfo && !isNaN(sInfo.start)) {
                    appendWindowStart = sInfo.start;
                }
                if (buffer.appendWindowEnd !== appendWindowEnd ||
      ↪ buffer.appendWindowStart !== appendWindowStart) {
                    buffer.appendWindowStart = 0;
                    buffer.appendWindowEnd = appendWindowEnd +
      ↪ APPEND_WINDOW_END_OFFSET;
                    buffer.appendWindowStart = Math.max(
      ↪ appendWindowStart - APPEND_WINDOW_START_OFFSET, 0);
                    logger.debug(`Updated append window for ${
      ↪ mediaInfo.type}. Set start to ${buffer.appendWindowStart}
      ↪ and end to ${buffer.appendWindowEnd}`);
                }

                resolve();
            } catch (e) {
                logger.warn(`Failed to set append window`);
                resolve();
            }
        });
    });
}
```

```
/* not refactored */
```

9	`netlify`	No	Cannot refactor because resolve is invoked asynchronously in an callback function
Original Source → Refactored Source Code			

Continued on next page

Table S5 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
async function fetchFiles(
    files: ImplementationFile[],
    readFile: ReadFile,
    readFileMetadata: ReadFileMetadata,
    apiName: string,
  ) {
    const sem = semaphore(MAX_CONCURRENT_DOWNLOADS);
    const promises = [] as Promise<ImplementationEntry | { error:
      ↪ boolean }>[];
    files.forEach(file => {
      promises.push(
        new Promise(resolve =>
          sem.take(async () => {
            try {
              const [data, fileMetadata] = await Promise.all([
                readFile(file.path, file.id, { parseText: true }),
                readFileMetadata(file.path, file.id),
              ]);
              resolve({ file: { ...file, ...fileMetadata }, data: data
    ↪  as string });
              sem.leave();
            } catch (error) {
              sem.leave();
              console.error(`failed to load file from ${apiName}: ${
    ↪ file.path}`);
              resolve({ error: true });
            }
          }),
        ),
      );
    });
    return Promise.all(promises).then(loadedEntries =>
      loadedEntries.filter(loadedEntry => !(loadedEntry as { error:
      ↪ boolean }).error),
    ) as Promise<ImplementationEntry[]>;
  }
```

```
/* not refactored */
```

10	fastify	Yes	Example in test code where new Promise() calls resolve in synchronous setting.

Original Source → Refactored Source Code

```
fastify.get('/return', opts, function (req, reply) {
    const promise = new Promise((resolve, reject) => {
      resolve({ hello: 'world' })
    })
    return promise
  })
```

```
fastify.get('/return', opts, function (req, reply) {
    const promise = Promise.resolve({ hello: 'world' });
    return promise
  })
```

Table S6: Pattern - reactionReturnsPromise

Number	Application	Refactored	Refactoring Comments
1	treeherder	Yes	Promise.resolve() returned in .catch; replace with return.

Original Source → Refactored Source Code

```
.catch((reason) =>
  Promise.resolve({
    result: {
      status: 'error',
      message_of_the_day:
        'Unable to connect to the https://mozilla-releng.net/
        ↪ treestatus API',
      reason: reason.toString(),
      tree: repoName,
    },
  }),
)
```

```
.catch((reason) => {
  return {
    result: {
      status: 'error',
      message_of_the_day:
        'Unable to connect to the https://mozilla-releng.net/
        ↪ treestatus API',
      reason: reason.toString(),
      tree: repoName,
    },
  };
})
```

2	treeherder	Yes	Return Promise.reject(e); throw e instead.

Original Source → Refactored Source Code

```
static getSeriesData(projectName, params) {
    return fetch(
      `${getProjectUrl(
        '/performance/data/',
        projectName,
      )}?${queryString.stringify(params)}`,
    ).then((resp) => {
      if (resp.ok) {
        return resp.json();
      }
      return Promise.reject('No series data found');
    });
  }
```

```
static getSeriesData(projectName, params) {
    return fetch(
      `${getProjectUrl(
        '/performance/data/',
        projectName,
      )}?${queryString.stringify(params)}`,
    ).then((resp) => {
      if (resp.ok) {
        return resp.json();
      }
      throw 'No series data found';
    });
  }
```

3	ui5-builder	Yes	Can simply return, instead of returning Promise.resolve().

Original Source → Refactored Source Code

```
depPromise.then(() => {
    if (projects[project.metadata.name]) {
            return Promise.resolve();
    }
    // details elided
}
```

```
depPromise.then(() => {
    if (projects[project.metadata.name]) {
            return;
    }
    // details elided
}
```

4	ui5-builder	Yes	This reaction returned a promise, but reactions already return promises! Removed outer promise, and had the code return, rather than resolve.

Original Source → Refactored Source Code

Continued on next page

Table S6 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			

```
.then((archiveContent) => new Promise((resolve) => {
    const zip = new yazl.ZipFile();
    const basePath = `/resources/${namespace}/`;
    archiveContent.forEach((content, path) => {
            if (!path.startsWith(basePath)) {
                    log.verbose(`Not bundling resource with path ${
      ↪ path} since it is not based on path ${basePath}`);
                    return;
            }
            // Remove base path. Absolute paths are not allowed in ZIP
      ↪  files
            const normalizedPath = path.replace(basePath, "");
            zip.addBuffer(content, normalizedPath);
    });
    zip.end();

    const pathPrefix = "/resources/" + namespace + "/";
    const res = resourceFactory.createResource({
            path: pathPrefix + bundleName,
            stream: zip.outputStream
    });
    resolve([res]);
}))
```

```
.then((archiveContent) => {
    const zip = new yazl.ZipFile();
    const basePath = `/resources/${namespace}/`;
    archiveContent.forEach((content, path) => {
            if (!path.startsWith(basePath)) {
                    log.verbose(`Not bundling resource with path ${
      ↪ path} since it is not based on path ${basePath}`);
                    return;
            }
            // Remove base path. Absolute paths are not allowed in ZIP
      ↪  files
            const normalizedPath = path.replace(basePath, "");
            zip.addBuffer(content, normalizedPath);
    });
    zip.end();

    const pathPrefix = "/resources/" + namespace + "/";
    const res = resourceFactory.createResource({
            path: pathPrefix + bundleName,
            stream: zip.outputStream
    });
    return [res];
});
```

5	appcenter-cli	No	This reaction does return a promise, but the promise is essentially a roundabout case of custom promisification: it sets up an error handler which resolves or rejects the promise, which is challenging to refactor.
Original Source → Refactored Source Code			

Continued on next page

Table S6 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
.then(() => {
    if (!sourcemapOutput) {
      // skip source map compose if source map is not enabled
      return;
    }

    const composeSourceMapsPath = getComposeSourceMapsPath();
    if (!composeSourceMapsPath) {
      throw new Error("react-native␣compose-source-maps.js␣scripts␣is␣
      ↪ not␣found");
    }

    const jsCompilerSourceMapFile = path.join(outputFolder, bundleName
      ↪  + ".hbc" + ".map");
    if (!fs.existsSync(jsCompilerSourceMapFile)) {
      throw new Error(`sourcemap file ${jsCompilerSourceMapFile} is
      ↪ not found`);
    }

    return new Promise((resolve, reject) => {
      const composeSourceMapsArgs = [sourcemapOutput,
      ↪ jsCompilerSourceMapFile, "-o", sourcemapOutput];

      // https://github.com/facebook/react-native/blob/master/react.
      ↪ gradle#L211
      // https://github.com/facebook/react-native/blob/master/scripts/
      ↪ react-native-xcode.sh#L178
      // packager.sourcemap.map + hbc.sourcemap.map = sourcemap.map
      const composeSourceMapsProcess = childProcess.spawn(
      ↪ composeSourceMapsPath, composeSourceMapsArgs);
      out.text(`${composeSourceMapsPath} ${composeSourceMapsArgs.join(
      ↪ "␣")}`);

      composeSourceMapsProcess.stdout.on("data", (data: Buffer) => {
        out.text(data.toString().trim());
      });

      composeSourceMapsProcess.stderr.on("data", (data: Buffer) => {
        console.error(data.toString().trim());
      });

      composeSourceMapsProcess.on("close", (exitCode: number) => {
        if (exitCode) {
          reject(new Error(`"compose-source-maps" command exited with
      ↪ code ${exitCode}.`));
        }

        // Delete the HBC sourceMap, otherwise it will be included in
      ↪ 'code-push' bundle as well
        fs.unlink(jsCompilerSourceMapFile, (err) => {
          if (err) {
            console.error(err);
            reject(err);
          }

          resolve(null);
        });
      });
    });
  });
```

```
/* code 5 here */
```

6	strapi	Yes	Promise.reject(e); replace with throw e

Original Source → Refactored Source Code

Continued on next page

Table S6 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
const validateComponentInput = data => {
    return yup
      .object({
        component: componentSchema,
        components: nestedComponentSchema,
      })
      .noUnknown()
      .validate(data, {
        strict: true,
        abortEarly: false,
      })
      .catch(error => Promise.reject(formatYupErrors(error)));
  };
```

```
const validateComponentInput = data => {
    return yup
      .object({
        component: componentSchema,
        components: nestedComponentSchema,
      })
      .noUnknown()
      .validate(data, {
        strict: true,
        abortEarly: false,
      })
      .catch(error => { throw formatYupErrors(error) });
  };
```

7	strapi	Yes	Promise.reject(e); replace with throw e

Original Source → Refactored Source Code

```
const validateUpdateComponentInput = data => {
    if (_.has(data, 'component')) {
      removeEmptyDefaults(data.component);
    }

    if (_.has(data, 'components') && Array.isArray(data.components)) {
      data.components.forEach(data => {
        if (_.has(data, 'uid')) {
          removeEmptyDefaults(data);
        }
      });
    }

    return yup
      .object({
        component: componentSchema,
        components: nestedComponentSchema,
      })
      .noUnknown()
      .validate(data, {
        strict: true,
        abortEarly: false,
      })
      .catch(error => Promise.reject(formatYupErrors(error)));
  };
```

```
const validateUpdateComponentInput = data => {
    if (_.has(data, 'component')) {
      removeEmptyDefaults(data.component);
    }

    if (_.has(data, 'components') && Array.isArray(data.components)) {
      data.components.forEach(data => {
        if (_.has(data, 'uid')) {
          removeEmptyDefaults(data);
        }
      });
    }

    return yup
      .object({
        component: componentSchema,
        components: nestedComponentSchema,
      })
      .noUnknown()
      .validate(data, {
        strict: true,
        abortEarly: false,
      })
      .catch(error => { throw formatYupErrors(error) });
  };
```

8	netlify-cms	Yes	Promise.reject(e); replace with throw e

Original Source → Refactored Source Code

```
parseResponse(response: Response) {
    const contentType = response.headers.get('Content-Type');
    if (contentType && contentType.match(/json/)) {
      return this.parseJsonResponse(response);
    }
    const textPromise = response.text().then(text => {
      if (!response.ok) {
        return Promise.reject(text);
      }
      return text;
    });
    return textPromise;
  }
```

```
parseResponse(response: Response) {
    const contentType = response.headers.get('Content-Type');
    if (contentType && contentType.match(/json/)) {
      return this.parseJsonResponse(response);
    }
    const textPromise = response.text().then(text => {
      if (!response.ok) {
        throw text;
      }
      return text;
    });
    return textPromise;
  }
```

9	netlify-cms	Yes	Promise.resolve(e); replace with return e

Original Source → Refactored Source Code

Continued on next page

Table S6 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
async entryExist(collection: Collection, path: string, slug: string,
      ↪ useWorkflow: boolean) {
    const unpublishedEntry =
      useWorkflow &&
      (await this.implementation
        .unpublishedEntry({ collection: collection.get('name'), slug
      ↪ })
        .catch(error => {
          if (error instanceof EditorialWorkflowError && error.
      ↪ notUnderEditorialWorkflow) {
            return Promise.resolve(false);
          }
          return Promise.reject(error);
        }));

    if (unpublishedEntry) return unpublishedEntry;

    const publishedEntry = await this.implementation
      .getEntry(path)
      .then(({ data }) => data)
      .catch(() => {
        return Promise.resolve(false);
      });

    return publishedEntry;
  }
```

```
async entryExist(collection: Collection, path: string, slug: string,
      ↪ useWorkflow: boolean) {
    const unpublishedEntry =
      useWorkflow &&
      (await this.implementation
        .unpublishedEntry({ collection: collection.get('name'), slug
      ↪ })
        .catch(error => {
          if (error instanceof EditorialWorkflowError && error.
      ↪ notUnderEditorialWorkflow) {
            return Promise.resolve(false);
          }
          return Promise.reject(error);
        }));

    if (unpublishedEntry) return unpublishedEntry;

    const publishedEntry = await this.implementation
      .getEntry(path)
      .then(({ data }) => data)
      .catch(() => {
        return false;
      });

    return publishedEntry;
  }
```

10	eleventy	Yes	Promise.reject(e); replace with throw e

Original Source → Refactored Source Code

```
this._generateTemplate(mapEntry, to).catch(function (e) {
    // Premature templateContent in layout render, this also happens
      ↪ in
    // TemplateMap.populateContentDataInMap for non-layout content
    if (EleventyErrorUtil.isPrematureTemplateContentError(e)) {
      usedTemplateContentTooEarlyMap.push(mapEntry);
    } else {
      return Promise.reject(
        new TemplateWriterWriteError(
          `Having trouble writing template: ${mapEntry.outputPath}`,
          e
        )
      );
    }
  })
```

```
this._generateTemplate(mapEntry, to).catch(function (e) {
    // Premature templateContent in layout render, this also happens
      ↪ in
    // TemplateMap.populateContentDataInMap for non-layout content
    if (EleventyErrorUtil.isPrematureTemplateContentError(e)) {
      usedTemplateContentTooEarlyMap.push(mapEntry);
    } else {
      throw new TemplateWriterWriteError(
          `Having trouble writing template: ${mapEntry.outputPath}`,
          e
        );
    }
  })
```

Table S7: Pattern - customPromisification

Number	Application	Refactored	Refactoring Comments
1	appcenter-cli	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

```
export function glob(pattern: string, options?: g.IOptions): Promise<
    ↪ string[]> {
  return new Promise<string[]>((resolve, reject) => {
    g(pattern, options, (err, matches) => {
      if (err) {
        reject(err);
      } else {
        resolve(matches);
      }
    });
  });
}
```

```
export function glob(pattern: string, options?: g.IOptions): Promise<
    ↪ string[]> {
  let g_promisified = util.promisify(g);
  return g_promisified(pattern, options);
}
```

Number	Application	Refactored	Refactoring Comments
2	eleventy	Yes	Needed Function.prototype.call to util.promisify'd function to set correct this.

Original Source → Refactored Source Code

```
return async function (data) {
    return new Promise(function (resolve, reject) {
      tmpl.render(data, function (err, res) {
        if (err) {
          reject(err);
        } else {
          resolve(res);
        }
      });
    });
  };
```

```
return async function (data) {
    let tmpl_render_promise = util.promisify(tmpl.render);
    return tmpl_render_promise.call(tmpl, data);
  };
```

Number	Application	Refactored	Refactoring Comments
3	ui5-builder	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

```
const findFiles = (folder) => {
    return new Promise((resolve, reject) => {
            recursive(folder, (err, files) => {
                    if (err) {
                            reject(err);
                    } else {
                            resolve(files);
                    }
            });
    });
};
```

```
const findFiles = (folder) => {
    let promisifiedRecursive = util.promisify(recursive);
    return promisifiedRecursive(folder);
};
```

Number	Application	Refactored	Refactoring Comments
4	ui5-builder	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

```
const findFiles = (folder) => {
    return new Promise((resolve, reject) => {
            recursive(folder, (err, files) => {
                    if (err) {
                            reject(err);
                    } else {
                            resolve(files);
                    }
            });
    });
};
```

```
const findFiles = (folder) => {
    let promisifiedRecursive = util.promisify(recursive);
    return promisifiedRecursive(folder);
};
```

Number	Application	Refactored	Refactoring Comments
5	ui5-builder	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

Continued on next page

Table S7 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
const findFiles = (folder) => {
    return new Promise((resolve, reject) => {
            recursive(folder, (err, files) => {
                    if (err) {
                            reject(err);
                    } else {
                            resolve(files);
                    }
            });
    });
};
```

```
const findFiles = (folder) => {
    let promisifiedRecursive = util.promisify(recursive);
    return promisifiedRecursive(folder);
};
```

6	mercurius	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

```
function sendRequest (request, url) {
    return function (opts) {
      return new Promise((resolve, reject) => {
        request({
          url,
          method: 'POST',
          body: opts.body,
          headers: {
            ...opts.headers,
            'content-type': 'application/json',
            'content-length': Buffer.byteLength(opts.body)
          },
          originalRequestHeaders: opts.originalRequestHeaders || {},
          context: opts.context
        }, (err, response) => {
          if (err) {
            return reject(err)
          }

          let data = ''
          response.stream.on('data', chunk => {
            data += chunk
          })

          eos(response.stream, (err) => {
            /* istanbul ignore if */
            if (err) {
              return reject(err)
            }

            try {
              const json = sJSON.parse(data.toString())

              if (json.errors && json.errors.length) {
                // return a `FederatedError` instance to keep `graphql
    ↪ ` happy
                // e.g. have something that derives from `Error`
                return reject(new FederatedError(json.errors))
              }

              resolve({
                statusCode: response.statusCode,
                json
              })
            } catch (e) {
              reject(e)
            }
          })
        })
      })
    }
  }
```

```
function sendRequest (request, url) {
    return function (opts) {
      const reqProm = util.promisify(request);
      return reqProm({
        url,
        method: 'POST',
        body: opts.body,
        headers: {
          ...opts.headers,
          'content-type': 'application/json',
          'content-length': Buffer.byteLength(opts.body)
        },
        originalRequestHeaders: opts.originalRequestHeaders || {},
        context: opts.context
      }).then(response => {
        let data = ''
        response.stream.on('data', chunk => {
          data += chunk
        })
        let eosProm = util.promisify(eos);
        return eosProm(response.stream).then(() => {
          try {
            const json = sJSON.parse(data.toString())

            if (json.errors && json.errors.length) {
              // return a `FederatedError` instance to keep `graphql`
    ↪ happy
              // e.g. have something that derives from `Error`
              throw new FederatedError(json.errors);
            }

            return {
              statusCode: response.statusCode,
              json
            };
          } catch (e) {
            throw e
          }
        }, err => {
          throw err;
        });
      }, err => {
        throw err;
      });
    }
  }
```

Continued on next page

Table S7 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			
7	mercurius	Yes	Complex, event-driven control flow prevented us from refactoring this code.

Original Source → Refactored Source Code

```
subscribe (topic, queue) {
    return new Promise((resolve, reject) => {
      function listener (value, cb) {
        queue.push(value.payload)
        cb()
      }

      const close = () => {
        this.emitter.removeListener(topic, listener)
      }

      this.emitter.on(topic, listener, (err) => {
        if (err) {
          return reject(err)
        }

        resolve()
      })
      queue.close = close
    })
}
```

```
// Not refactored.
```

8	mercurius	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

```
publish (event) {
    return new Promise((resolve, reject) => {
      this.pubsub.publish(event, (err) => {
        if (err) {
          return reject(err)
        }
        resolve()
      })
    }).catch(err => {
      this.fastify.log.error(err)
    })
}
```

```
publish (event) {
    console.log('CE␣--␣ihp␣--␣3');
    let thisPubsubPublish = util.promisify(this.pubsub.publish);
    return thisPubsubPublish.call(this.pubsub, event).then(() => {
      return
    }).catch(err => {
      this.fastify.log.error(err)
    });
  }
```

9	Boostnote	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

```
function saveToFile(data, filename) {
    return new Promise((resolve, reject) => {
      fs.writeFile(filename, data, err => {
        if (err) return reject(err)

        resolve(filename)
      })
    })
  }
```

```
function saveToFile(data, filename) {
    const writeFileProm = util.promisify(fs.writeFile);
    return writeFileProm(filename, data);
}
```

10	appcenter-cli	Yes	Simple call to util.promisify.

Original Source → Refactored Source Code

```
const files = await new Promise<string[]>((resolve, reject) => {
    glob(pattern, (err, matches) => {
      if (err) {
        reject(err);
      } else {
        resolve(matches);
      }
    });
});
```

```
const files = await new Promise<string[]>((resolve, reject) => {
    const globPromisified = util.promisify(glob);
    resolve(globPromisified(pattern));
});
```

Table S8: Pattern - explicitPromiseConstructor

Number	Application	Refactored	Refactoring Comments
1	Boostnote	Yes	Here, we had to promisify writeFile. We could do this with util.promisify, but the fs library comes with promise-based versions of their APIs (thanks to fs.promises). This refactoring is possible because fetchSnippet already returns a promise.

Original Source → Refactored Source Code

```
function createSnippet(snippetFile) {
  return new Promise((resolve, reject) => {
    const newSnippet = {
      id: crypto.randomBytes(16).toString('hex'),
      name: 'Unnamed␣snippet',
      prefix: [],
      content: '',
      linesHighlighted: []
    }
    fetchSnippet(null, snippetFile)
      .then(snippets => {
        snippets.push(newSnippet)
        fs.writeFile(
          snippetFile || consts.SNIPPET_FILE,
          JSON.stringify(snippets, null, 4),
          err => {
            if (err) reject(err)
            resolve(newSnippet)
          }
        )
      })
      .catch(err => {
        reject(err)
      })
  })
}
```

```
function createSnippet(snippetFile) {
  return fetchSnippet(null, snippetFile)
    .then(snippets => {
      const newSnippet = {
        id: crypto.randomBytes(16).toString('hex'),
        name: 'Unnamed␣snippet',
        prefix: [],
        content: '',
        linesHighlighted: []
      }
      snippets.push(newSnippet)
      return fs.promises.writeFile(
        snippetFile || consts.SNIPPET_FILE,
        JSON.stringify(snippets, null, 4))
        .then(() => newSnippet)
        .catch(err => { throw err })
    })
}
```

2	Boostnote	Yes	We similarly had to promisify writeFile. This refactoring is possible because fetchSnippet already returns a promise.

Original Source → Refactored Source Code

```
function deleteSnippet(snippet, snippetFile) {
    return new Promise((resolve, reject) => {
        fetchSnippet(null, snippetFile).then(snippets => {
        snippets = snippets.filter(
            currentSnippet => currentSnippet.id !== snippet.id
        )
        fs.writeFile(
            snippetFile || consts.SNIPPET_FILE,
            JSON.stringify(snippets, null, 4),
            err => {
                if (err) reject(err)
                resolve(snippet)
            }
        )
        })
    })
}
```

```
function deleteSnippet(snippet, snippetFile) {
    return fetchSnippet(null, snippetFile).then(snippets => {
        snippets = snippets.filter(
          currentSnippet => currentSnippet.id !== snippet.id
        )
        return fs.promises.writeFile(
            snippetFile || consts.SNIPPET_FILE,
            JSON.stringify(snippets, null, 4)).then(err => {
                if (err) throw err
                return snippet
        })
    })
}
```

3	Boostnote	No	This code sets up complex control flow through a request handler. Control flow passes up through a callback, into the original promise—we did not feel comfortable refactoring this code.

Original Source → Refactored Source Code

Continued on next page

Table S8 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments
Original Source → Refactored Source Code			

```
return new Promise((resolve, reject) => {
    const td = createTurndownService()

    // ...

    const req = request.request(url, res => {
        let data = ''

        res.on('data', chunk => {
            data += chunk
        })

        res.on('end', () => {
            const markdownHTML = td.turndown(data)

            if (dispatch !== null) {
                createNote(storage, {
                    type: 'MARKDOWN_NOTE',
                    folder: folder,
                    title: '',
                    content: markdownHTML
                }).then(note => {
                    // ...
                    resolve({ result: true, error: null })
                })
            } else {
                createNote(storage, {
                    /* ... */
                }).then(note => {
                    resolve({ result: true, note, error: null })
                })
            }
        })
    })

    req.on('error', e => {
        console.error('error in parsing URL', e)
        reject({
            result: false,
            error: ERROR_MESSAGES[e.code] || ERROR_MESSAGES.UNEXPECTED
        })
    })

    req.end()
})
```

```
//
// We did not refactor this instance.
//
```

4	dash.js	Yes	The function used to return a promise which resolved with a promise—we removed the outer promise, and returned the promise that was returned originally.

Original Source → Refactored Source Code

Continued on next page

Table S8 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
function _updateRepresentation(currentRep) {
    return new Promise((resolve, reject) => {
        const hasInitialization = currentRep.hasInitialization();
        const hasSegments = currentRep.hasSegments();

        // If representation has initialization and segments
      ↪ information we are done
        // otherwise, it means that a request has to be made to get
      ↪ initialization and/or segments information
        const promises = [];

        promises.push(segmentsController.updateInitData(currentRep,
      ↪ hasInitialization));
        promises.push(segmentsController.updateSegmentData(currentRep,
      ↪  hasSegments));

        Promise.all(promises)
            .then((data) => {
                if (data[0] && !data[0].error) {
                    currentRep = _onInitLoaded(currentRep, data[0]);
                }
                if (data[1] && !data[1].error) {
                    currentRep = _onSegmentsLoaded(currentRep, data
      ↪ [1]);
                }
                _setMediaFinishedInformation(currentRep);
                _onRepresentationUpdated(currentRep);
                resolve();
            })
            .catch((e) => {
                reject(e);
            });
    });
}
```

```
function _updateRepresentation(currentRep) {
    const hasInitialization = currentRep.hasInitialization();
    const hasSegments = currentRep.hasSegments();

    // If representation has initialization and segments information
      ↪ we are done
    // otherwise, it means that a request has to be made to get
      ↪ initialization and/or segments information
    const promises = [];

    promises.push(segmentsController.updateInitData(currentRep,
      ↪ hasInitialization));
    promises.push(segmentsController.updateSegmentData(currentRep,
      ↪ hasSegments));

    return Promise.all(promises)
        .then((data) => {
            if (data[0] && !data[0].error) {
                currentRep = _onInitLoaded(currentRep, data[0]);
            }
            if (data[1] && !data[1].error) {
                currentRep = _onSegmentsLoaded(currentRep, data[1]);
            }
            _setMediaFinishedInformation(currentRep);
            _onRepresentationUpdated(currentRep);
            return;
        })
        .catch((e) => {
            throw e;
        });
}
```

5	dash.js	Yes	Similar to the previous case, the function returned a promise which returned a promise, and we removed the outer one.

Original Source → Refactored Source Code

```
function startPreloading(mediaSource, previousBuffers) {
    return new Promise((resolve, reject) => {

        if (getPreloaded()) {
            reject();
            return;
        }

        logger.info(`[startPreloading] Preloading next stream with id
      ↪ ${getId()}`);
        setPreloaded(true);

        _commonMediaInitialization(mediaSource, previousBuffers)
            .then(() => {
                for (let i = 0; i < streamProcessors.length &&
      ↪ streamProcessors[i]; i++) {
                    streamProcessors[i].setExplicitBufferingTime(
      ↪ getStartTime());
                    streamProcessors[i].getScheduleController().
      ↪ startScheduleTimer();
                }
                resolve();
            })
            .catch(() => {
                setPreloaded(false);
                reject();
            });
    });
}
```

```
function startPreloading(mediaSource, previousBuffers) {

    if (getPreloaded()) {
        reject();
        return Promise.reject();
    }

    logger.info(`[startPreloading] Preloading next stream with id ${
      ↪ getId()}`);
    setPreloaded(true);

    return _commonMediaInitialization(mediaSource, previousBuffers)
        .then(() => {
            for (let i = 0; i < streamProcessors.length &&
      ↪ streamProcessors[i]; i++) {
                streamProcessors[i].setExplicitBufferingTime(
      ↪ getStartTime());
                streamProcessors[i].getScheduleController().
      ↪ startScheduleTimer();
            }
            return;
        })
        .catch(() => {
            setPreloaded(false);
            throw undefined;
        });
}
```

Table S9: Pattern - explicitPromiseConstructor

Number	Application	Refactored	Refactoring Comments
6	dash.js	No	We felt that we should be able to refactor this case, but refactoring introduced synchrony that caused a failing test.

Original Source → Refactored Source Code

```
function _commonMediaInitialization(mediaSource, previousBufferSinks)
    ↪ {
    return new Promise((resolve, reject) => {
        checkConfig();

        isUpdating = true;
        addInlineEvents();


        let element = videoModel.getElement();

        MEDIA_TYPES.forEach((mediaType) => {
            if (mediaType !== Constants.VIDEO || (!element || (element
    ↪  && (/^VIDEO$/i).test(element.nodeName)))) {
                _initializeMediaForType(mediaType, mediaSource);
            }
        });

        _createBufferSinks(previousBufferSinks)
            .then((bufferSinks) => {
                isUpdating = false;

                if (streamProcessors.length === 0) {
                    const msg = 'No␣streams␣to␣play.';
                    errHandler.error(new DashJSError(Errors.
    ↪ MANIFEST_ERROR_ID_NOSTREAMS_CODE, msg, manifestModel.
    ↪ getValue()));
                    logger.fatal(msg);
                } else {
                    _checkIfInitializationCompleted();
                }

                // All mediaInfos for texttracks are added to the
    ↪ TextSourceBuffer by now. We can start creating the tracks
                textController.createTracks(streamInfo);

                resolve(bufferSinks);
            })
            .catch((e) => {
                reject(e);
            });
    });

}
```

```
//
// We did not refactor this instance.
//
```

7	ui5-builder	Yes	Again, a promise within a promise, and we removed the outer promise.

Original Source → Refactored Source Code

```
promises.push(new Promise((resolve) => {
    return this._pool.getModuleInfo(info.name).then((moduleInfo) => {
            if (moduleInfo.name) {
                info.module = moduleInfo.name;
            }
            resolve();
    });
}));
```

```
promises.push(this._pool.getModuleInfo(info.name).then((moduleInfo) =>
    ↪ {
    if (moduleInfo.name) {
            info.module = moduleInfo.name;
    }
    return;
}));
```

8	ui5-builder	Yes	The promise returned by the function was very busy, and was doing roundabout custom promisification. We explicitly promisified the function call, and returned that with error handling mimicking the logic of the original.

Original Source → Refactored Source Code

Continued on next page

Table S9 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
_analyze(xml, info, isFragment) {
  // ...

  return new Promise( (resolve, reject) => {
    this._parser.parseString(xml, (err, result) => {
      // parse error
      if ( err ) {
        this.busy = false;
        reject(new Error(`Error while parsing XML document ${info.name
        ↪ }: ${err.message}`));
        return;
      }

      if ( !result ) {
        // Handle empty xml views/fragments
        reject(new Error("Empty XML Document."));
        return;
      }

      if ( isFragment ) {
        // all fragments implicitly depend on the fragment class
        this.info.addImplicitDependency(FRAGMENT_MODULE);
        this._analyzeNode(result);
      } else {
        // views require a special handling of the root node
        this._analyzeViewRootNode(result);
      }

      Promise.all(this.promises).then( () => {
        this.busy = false;
        resolve(info);
      });
    });
  });
}
```

```
_analyze(xml, info, isFragment) {
  // ...

  let promisified = util.promisify(this._parser.parseString);
  return promisified(xml).then( result => {
    if (!result)
      throw new Error("Empty XML Document.");

    if ( isFragment ) {
      this.info.addImplicitDependency(FRAGMENT_MODULE);
      this._analyzeNode(result);
    } else {
      this._analyzeViewRootNode(result);
    }

    return Promise.all(this.promises).then( () => {
      this.busy = false;
      return info;
    });
  }, err => {
    this.busy = false;
    throw new Error(`Error while parsing XML document ${info.name}: ${
      ↪ err.message}`);
  });
}
```

9	appcenter-cli	Yes	Another instance where custom promisification and the explicit promise constructor co-occur.

Original Source → Refactored Source Code

```
function createFakeAppFile(appFilePath: string, entryNames: string[]):
    ↪ Promise<string> {
  return new Promise<string>((resolve, reject) => {
    temp.mkdir("app-validator-tests", (error, dirPath) => {
      try {
        if (error) {
          reject(error);
          return;
        }

        const inputFile = path.join(dirPath, appFilePath);
        const zip = new JsZip();
        for (const entryName of entryNames) {
          zip.file(entryName, Buffer.from("Fake file"));
        }

        JsZipHelper.writeZipToPath(inputFile, zip, "STORE").then(
          () => resolve(inputFile),
          (writingError) => reject(writingError)
        );
      } catch (err) {
        reject(err);
      }
    });
  });
}
```

```
function createFakeAppFile(appFilePath: string, entryNames: string[]):
    ↪ Promise<string> {
  let tempMkdirPromise = util.promisify(temp.mkdir);
  return tempMkdirPromise("app-validator-tests").then((dirPath : any
    ↪ ) => {
    const inputFile = path.join(dirPath, appFilePath);
    const zip = new JsZip();
    for (const entryName of entryNames) {
      zip.file(entryName, Buffer.from("Fake file"));
    }

    return JsZipHelper.writeZipToPath(inputFile, zip, "STORE").then(
      () => inputFile,
      (writingError : any) => { throw writingError; }
    );
  }).catch((err : any) => {
    throw err;
  });
}
```

10	strapi	No	Any refactorings involving setTimeout are fraught.

Original Source → Refactored Source Code

Continued on next page

Table S9 – *Continued from previous page*

Number	Application	Refactored	Refactoring Comments

Original Source → Refactored Source Code

```
const initialize = middlewareKey => {
    if (this.middleware[middlewareKey].loaded === true) return;

    const module = this.middleware[middlewareKey].load;

    return new Promise((resolve, reject) => {
      const timeout = setTimeout(
        () => reject(`(middleware: ${middlewareKey}) is taking too
      ↪ long to load.`),
        middlewareConfig.timeout || 1000
      );

      this.middleware[middlewareKey] = merge(this.middleware[
      ↪ middlewareKey], module);

      Promise.resolve()
        .then(() => module.initialize())
        .then(() => {
          clearTimeout(timeout);
          this.middleware[middlewareKey].loaded = true;
          resolve();
        })
        .catch(err => {
          clearTimeout(timeout);

          if (err) {
            return reject(err);
          }
        });
    });
  };
```

```
//
// We did not refactor this instance.
//
```

Appendix B

Database Usage Optimizations

This appendix contains complete information for the experiments conducted in Chapter 5. First, links to the code corresponding to each HTTP Request ID are given in Table S1.

B.1 Raw Data

RQ3: How do the refactorings affect performance?

The first part of this research question, wherein the performance difference of all refactoring opportunities was examined, is presented first. Each column corresponds to the "HTTP Request ID" from the graph in the chapter. 10 run times were gathered before (first 10 rows), and after (last 10 rows) refactoring. The graph in the chapter reports means and standard deviations for these data sets. The raw numbers are given in the black cells.

The next part of the research question examines how the refactoring opportunities scale. Five sets of results are provided corresponding to each HTTP request being studied. Three database scales are investigated for each. The raw numbers are in the black cells, and aggregates are computed throughout.

ID	0	1	2	3	4	5
Run 1 Before	296.5422087	342.3916879	439.6706128	660.5764589	379.8503571	236.455646
Run 2 Before	297.2173848	334.0338912	436.371685	514.7209811	357.3590293	260.0362577
Run 3 Before	296.7770619	319.0932541	427.3306279	625.0453887	345.5173397	236.075304
Run 4 Before	306.6585417	349.8703504	407.0242591	606.6610117	377.5784178	251.6018581
Run 5 Before	327.048172	350.4867721	429.1028571	541.465981	370.9511352	252.0939388
Run 6 Before	319.0272932	353.3099599	422.9840903	561.8722239	350.3729053	258.1072946
Run 7 Before	301.8668461	365.006474	423.3101697	546.3098879	356.7801609	241.9330091
Run 8 Before	310.4705081	353.9094543	435.9675951	607.7707334	357.9924269	240.0164981
Run 9 Before	285.3387799	352.1118441	411.8937869	560.5535321	343.7547359	239.818984
Run 10 Before	296.5833168	331.2777181	426.4606433	544.488255	358.2252998	235.3724079
Run 1 After	112.6107941	116.0074501	262.6131229	411.7019057	111.3897181	131.9535041
Run 2 After	112.0591736	115.8520789	221.1474237	392.8332429	124.4094291	116.2680387
Run 3 After	100.94871	114.7537441	224.8161006	306.5939159	115.9604883	110.7396469
Run 4 After	102.5760531	166.4302011	225.8762507	347.4592581	113.1825919	121.155684
Run 5 After	109.465014	112.3772321	237.2390442	358.220077	163.2544618	100.02877
Run 6 After	97.59109831	123.9580231	218.9853449	338.1895022	126.7860384	101.9255929
Run 7 After	98.79903793	127.8616757	267.5582781	352.95013	126.0931506	95.06175566
Run 8 After	95.76102209	117.886333	223.2188554	312.6392736	121.0590048	93.60310078
Run 9 After	114.1138463	114.7114372	229.1717229	311.760932	120.9552608	99.40444517
Run 10 After	98.45451641	118.4825902	215.3943319	321.3568254	112.1688318	100.329926

6	7	8	9	10	11	12
255.5270448	31.54890823	37.47572422	30.19251537	383.3687129	84.24210215	84.52796698
261.5263901	29.17892075	38.50778913	29.73556089	268.8404732	64.59138489	67.06114769
254.4934812	29.24011087	39.18856335	31.11564922	273.0344238	62.55630398	63.55715609
320.1583767	28.79978704	49.51084423	29.23689079	271.2723522	56.82597017	60.92186975
257.5286303	28.17663002	38.37754869	30.2588501	266.7931714	59.99328184	64.33243656
257.7469811	29.06025362	39.0714612	28.90237379	260.5706091	63.15875578	62.72778606
239.7154741	31.41610718	37.35225487	28.83481503	279.4610162	48.51346397	56.35801697
273.9802361	28.87679386	31.90088511	32.19949436	249.8038549	62.25871897	59.65124035
248.1825027	29.27708387	59.0689683	29.44326591	280.0218821	61.4637866	59.45785904
258.1743531	25.97849178	35.62498426	29.12376213	264.6863651	61.11528063	73.93431377
86.45106888	24.3463006	14.71561813	23.68372011	37.87574339	30.14804268	29.4408083
73.05752563	22.33225107	21.37562227	22.43517399	34.91607189	26.09172201	27.69058418
79.25007915	22.87772655	21.58984804	32.12573624	37.09590673	29.67933273	37.11771631
65.5976429	23.76235294	21.38733006	24.30892897	30.95006895	33.36138439	41.78159809
71.08857393	21.40966797	21.51010609	25.71847105	35.3449707	27.21611023	21.80800962
75.41383028	20.39828777	22.70023298	25.79724407	33.05799627	43.15440416	44.62874699
71.54601097	22.014503	22.70336914	27.41158867	47.20662498	35.04477406	42.24501705
67.7587328	22.20355415	21.66755199	25.67760611	41.8272953	26.51093817	30.12857103
63.52026463	21.23545408	22.39611387	24.03600502	34.3089633	27.40306807	25.13212967
71.86879301	25.27088881	22.06269693	26.00050211	32.25929689	33.27373123	28.15039921

13	14	15	16	17	18	19
37.39189816	33.11767292	10.63178492	54.81344128	84.93616056	82.35615921	99.73111868
68.1372509	32.97045374	7.013574123	53.68588591	83.27398014	94.85325432	84.99699163
47.01346493	33.56807327	8.029529095	51.6089592	85.24731398	85.550354	75.72969532
56.18071604	34.47556877	7.581944942	55.25237226	84.87770891	81.72766399	92.80006981
50.03034306	33.82006216	6.800681114	49.55034685	84.49906063	81.47541761	91.03131199
53.88670588	34.12095976	7.817266941	55.33009958	89.28994942	83.76673985	84.85094976
48.11879206	34.04524374	8.438535213	48.34311914	86.20543003	85.07974434	77.52989578
45.26379204	33.96441031	7.557134151	54.3865428	86.5872016	81.72254133	84.8077879
71.28859806	34.37111568	7.022022724	55.97233963	83.40580368	88.55902386	87.66806698
52.94946194	33.76622105	7.312335968	57.21450329	83.69869423	85.61793804	78.61269522
36.44192076	23.38076782	5.471279621	49.95228624	82.55388451	81.14607239	79.05474329
32.68654299	23.54316568	5.415356159	52.86624002	83.79175138	81.44713211	87.32302809
41.93791676	25.95669317	6.261838913	42.5189333	80.33034897	79.65381527	80.80437183
33.59411812	23.65199232	5.902676105	46.0874753	80.88407183	76.12237024	71.12371016
30.72180223	24.8035841	5.860949993	43.84358025	81.29878283	78.69846201	81.22189283
41.33065605	23.92762995	5.831378937	49.11858511	71.89520502	78.32153988	71.29275322
31.54559374	23.83484125	5.433281898	45.17965269	71.80862808	77.98308516	79.14591217
32.82648802	24.04220581	5.7937603	44.12644291	68.60209799	81.00055885	69.80568314
38.26809692	24.97240496	5.546839714	48.06220913	81.89066124	81.23598766	80.09354782
40.37186003	24.94411182	5.286148071	47.95165491	79.13215113	80.9926548	80.02868891

20	21	22	23	24	25	26
178.1560512	54.28825903	70.68257904	61.35862303	60.25461388	53.20445919	48.54474258
121.1437821	60.8412118	57.92955065	73.34398413	56.83326101	52.971591	40.76281834
128.681201	54.51878023	61.09590721	60.37153864	48.49744701	61.51666164	49.47641563
112.902308	189.8686132	52.36235094	56.59376431	61.45610094	67.48612309	37.97711182
130.3009109	98.30597734	60.34437561	60.05402899	56.18875599	69.4939661	43.04938984
118.6056561	66.85883713	52.76358604	60.23706007	49.76257133	68.20813465	72.60159969
115.0580769	62.03088808	55.48856926	68.86966896	53.71933794	74.83953285	59.77196217
110.9197898	72.85479307	52.38694	62.49248075	57.24282932	64.37456131	68.42672062
106.5751801	65.01906538	51.21576691	54.24013519	53.40640879	104.3642292	48.74887562
112.6409097	56.1626091	56.06521177	59.30028296	56.45391417	55.14083004	46.08584309
78.13118124	25.08976984	33.39959431	39.3816328	31.95464516	53.74621773	46.96408367
74.44669914	18.67787838	32.80821991	27.38890696	29.48304319	52.39632511	38.30247784
66.62766504	23.19646025	30.37513208	36.63254499	30.83614302	51.41365051	40.47464466
51.93525219	18.87626886	31.69210482	42.38354683	31.04445314	72.07255459	36.94680119
62.63871002	22.23526621	24.90387011	41.90424013	32.92012787	51.56147957	36.10220051
72.67511606	21.42924166	29.44344616	29.09028769	28.53861332	55.80633354	41.38393879
73.01624012	21.96119881	23.98497677	30.60761595	31.15051413	49.91593075	37.88137054
67.67755175	19.22198296	27.42753363	33.70332384	30.52470684	52.11841393	37.9589262
68.408494	24.62135696	23.77592993	33.4468298	29.65719795	62.46364975	39.71023655
68.89422512	18.05610418	27.86280394	31.35187101	29.34483385	50.41362858	39.12059021

Project	HTTP Request ID	Link
youtube-clone	0	link [244]
youtube-clone	1	link [245]
youtube-clone	2	link [246]
youtube-clone	3	link [247]
youtube-clone	4	link [248]
youtube-clone	5	link [249]
youtube-clone	6	link [250]
eventbright	7	link [80]
eventbright	8	link [81]
eventbright	9	link [82]
eventbright	10	link [83]
eventbright	11	link [84]
eventbright	12	link [85]
eventbright	13	link [86]
employee-tracker	14	link [78]
employee-tracker	15	link [79]
Graceshopper-Elektra	16	link [104]
Math_Fluency_App	17	link [145]
Math_Fluency_App	18	link [146]
Math_Fluency_App	19	link [147]
property-manage	20	link [178]
NetSteam	21	link [162]
NetSteam	22	link [163]
NetSteam	23	link [164]
NetSteam	24	link [165]
wall	25	link [228]
wall	26	link [229]

Table S1: HTTP Request ID Code Mappings

		Scale = 10 (Before)		Scale = 10 (After)		Scale = 100 (Before)		Scale = 100 (After)		Scale = 1000 (Before)		Scale = 1000 (After)	
Application	Link to Fn Under Test	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev
employee-tracker	viewEmployees	57.15258532	2.26226939	34.32257767	3.319099893	374.7352178	19.83148014	153.9784529	5.646152874	2495.929026	105.3503482	1010.470114	59.82907639
		56.08637714		30.24459362		354.5324516		152.8123074		2422.508628		903.4403372	
	Perf Factor 10	56.80902481		34.00312424		395.461688		153.270895		2768.550002		1008.342466	
1.665160055		62.07994938		30.64050102		392.6381264		159.4298534		2415.134962		1002.464498	
		54.68844128		36.34280872		363.3562689		155.3109932		2439.126434		991.7239799	
	Perf Factor 100	57.45268822		34.44225025		333.8408365		157.1890478		2433.156911		925.33459	
2.433686082		55.91737366		33.72490883		375.5795937		159.5867023		2449.278038		1018.92368	
		54.32898808		42.13974094		376.848156		151.0016356		2512.663528		1084.713685	
	Perf Factor 1000	56.80100632		34.13665199		380.5106592		153.8688593		2477.418863		1079.240101	
2.470067142		58.4928875		34.92978191		397.9133177		140.1046219		2485.711101		1031.864879	
		58.86911678		32.62141514		376.6710796		157.2096128		2555.741795		1058.652921	
eventbright	getEvents4	111.3764592	12.27972081	31.93800249	6.445866555	797.3474434	59.6995168	49.52956829	3.342988163	7001.482782	327.2739117	214.6137006	34.19465845
		111.6727104		23.41290951		897.7271814		47.36740875		6756.523119		196.8082781	
	Perf Factor 10	121.9671812		28.53225327		811.6809139		45.62192726		6717.610682		205.0658092	
3.487270666		117.4528599		36.25894928		882.9317036		46.06086922		7572.294622		211.6975899	
		105.7877245		31.02360439		695.0254841		51.14043903		6756.973058		210.0827236	
	Perf Factor 100	86.90520191		41.11800766		799.5384426		51.5728178		6952.402718		205.4706707	
16.09841295		105.817009		25.49963474		746.8349237		46.35743141		7059.229768		202.157588	
		111.1030636		39.85428429		803.0140829		53.73471928		7478.30205		303.9714079	
	Perf Factor 1000	127.2731037		25.73008823		764.1296511		47.31891918		7248.967765		177.0932636	
32.62365246		124.925931		38.15420246		791.9611063		51.60024166		6699.241446		201.9867306	
		100.859807		29.79609108		780.6309443		54.52090931		6773.282589		231.8029442	
property-manage	getProperties	56.90932913	9.531586852	33.70714617	4.207218461	246.0574474	23.67101414	111.0540639	4.848989794	1333.635854	76.6832396	786.4386488	32.1491564
		54.60064602		27.05339336		229.9760456		117.7619658		1211.464869		782.7760344	
	Perf Factor 10	51.4209671		33.16272259		211.7035236		115.2461777		1285.810678		739.3372889	
1.68834611		49.64558411		31.62069893		240.3335018		106.370266		1238.919923		780.0881462	
		81.32437897		35.95491314		249.6863194		111.3240986		1347.678996		779.4366665	
	Perf Factor 100	49.66529655		37.73286057		302.1370163		113.8987141		1395.609416		812.6235361	
2.215654599		60.70725155		35.05593872		237.0385818		112.2086439		1415.996384		838.0991144	
		50.98642159		38.96317101		260.3886957		112.8055725		1308.922302		742.93573	
	Perf Factor 1000	61.01507187		35.77504253		237.5855112		113.4886856		1440.91305		771.1791363	
1.695791345		53.11094093		35.3135519		240.6365461		104.4201488		1303.593621		824.5596857	
		56.6167326		26.43916893		251.0887327		103.016366		1387.449304		793.3511496	
NetSteam	getReviewsForVid	77.05017395	14.72160379	39.00994816	9.172634262	337.6645753	81.77039376	41.62413406	6.119287515	2129.344193	161.8408128	108.0597514	12.90249138
		55.35832405		36.49987602		332.4349146		39.01986217		2136.904094		124.0966644	
	Perf Factor 10	100.835494		32.45882702		519.9663191		41.06946564		2239.967981		111.5521526	
1.975141665		75.66722584		40.86222076		269.2389965		35.65230656		2460.963181		87.86075687	
		69.98494625		42.66977978		353.4762383		39.2476387		2137.441291		108.635087	
	Perf Factor 100	98.25138474		30.76279068		361.3187485		31.76107883		1809.50442		100.2336445	
8.11223063		59.73704243		36.73888588		229.3985672		43.55233097		2031.759841		91.6466608	
		83.05284691		27.13776684		280.2233133		39.37084675		2096.688244		106.4584732	
	Perf Factor 1000	71.1099062		35.36546898		347.5319557		52.21240997		2132.039043		112.0710535	
19.70524793		73.83765507		50.18377209		397.5482674		45.08726025		2109.063425		129.7673531	
		82.66691399		57.42009354		285.5084324		49.26814079		2139.110408		108.2756681	
youtube-clone	searchUser	360.3031902	20.95403345	118.0600576	7.118029594	1937.424227	36.84084409	152.9622237	10.24310423	18171.8628	968.6944864	471.0747257	252.9187266
		354.6113157		116.2387857		1998.543572		146.8941793		17687.24106		1177.905577	
	Perf Factor 10	399.9294786		108.682826		1967.930428		168.1511545		18411.40728		366.9729233	
3.051863581		334.3724871		131.3627262		1913.916061		141.2384701		17866.41601		355.8913021	
		351.0997753		118.9472589		1910.133828		148.6189346		17708.71675		362.1227856	
	Perf Factor 100	366.7486725		117.9075928		1895.572589		140.4665251		17897.74838		354.1614819	
12.66603073		356.8432655		111.736908		1989.267398		144.8240299		20865.91642		433.6041231	
		390.8570023		123.5586615		1909.695372		153.3905878		17923.81666		454.5385752	
	Perf Factor 1000	340.8059597		119.8625383		1929.934538		156.2924147		17811.13509		361.7325869	
38.57532958		345.4486322		108.8637619		1950.429159		163.6325703		17808.927		358.7040863	
		362.3153133		123.439517		1908.819321		166.1133709		17737.30336		485.1138163	

RQ5: What is the running time of REFORMULATOR?

The next pages show the raw run times gathered from: installing projects, building QLDBs, and running the query. Times were gathered using the Unix `time` command; `usr` time corresponds to CPU time, `sys` time corresponds to system CPU cycles originating from user code, and `real` time corresponds to the wall clock time. Note that QLDB and query run times take advantage of multiple cores, while npm install time does not.

employee-tracker			eventbright		
Install	QLDB	Query Time	Install	QLDB	Query Time
real 0m4.346s	real 0m6.798s	real 0m5.061s	real 0m8.645s	real 0m8.199s	real 0m8.471s
user 0m2.826s	user 0m21.630s	user 0m27.364s	user 0m10.485s	user 0m27.181s	user 0m31.262s
sys 0m1.686s	sys 0m1.633s	sys 0m1.303s	sys 0m1.165s	sys 0m2.105s	sys 0m1.421s
real 0m1.958s	real 0m7.177s	real 0m5.218s	real 0m8.960s	real 0m8.601s	real 0m8.369s
user 0m2.424s	user 0m22.571s	user 0m28.223s	user 0m9.947s	user 0m26.348s	user 0m31.176s
sys 0m1.521s	sys 0m1.646s	sys 0m1.428s	sys 0m1.215s	sys 0m2.436s	sys 0m1.458s
real 0m2.022s	real 0m7.018s	real 0m5.088s	real 0m8.526s	real 0m8.319s	real 0m8.439s
user 0m2.526s	user 0m21.296s	user 0m28.748s	user 0m10.269s	user 0m27.932s	user 0m31.474s
sys 0m1.602s	sys 0m1.590s	sys 0m1.497s	sys 0m1.274s	sys 0m2.140s	sys 0m1.364s
real 0m1.910s	real 0m6.969s	real 0m5.289s	real 0m8.056s	real 0m8.297s	real 0m8.397s
user 0m2.308s	user 0m22.061s	user 0m28.662s	user 0m9.531s	user 0m25.566s	user 0m30.251s
sys 0m1.546s	sys 0m1.688s	sys 0m1.652s	sys 0m1.147s	sys 0m2.226s	sys 0m1.405s
real 0m2.018s	real 0m7.239s	real 0m5.270s	real 0m8.580s	real 0m8.305s	real 0m8.356s
user 0m2.462s	user 0m21.817s	user 0m28.981s	user 0m10.225s	user 0m26.533s	user 0m30.848s
sys 0m1.724s	sys 0m1.853s	sys 0m1.600s	sys 0m1.244s	sys 0m2.268s	sys 0m1.340s
real 0m2.237s	real 0m7.128s	real 0m5.002s	real 0m8.641s	real 0m8.266s	real 0m8.447s
user 0m2.821s	user 0m21.039s	user 0m27.109s	user 0m10.301s	user 0m26.792s	user 0m31.909s
sys 0m1.733s	sys 0m1.835s	sys 0m1.436s	sys 0m1.354s	sys 0m2.132s	sys 0m1.312s
real 0m1.886s	real 0m7.179s	real 0m5.217s	real 0m8.719s	real 0m8.166s	real 0m8.191s
user 0m2.341s	user 0m22.896s	user 0m29.180s	user 0m9.987s	user 0m26.910s	user 0m28.744s
sys 0m1.560s	sys 0m1.863s	sys 0m1.476s	sys 0m1.357s	sys 0m2.008s	sys 0m1.378s
real 0m2.193s	real 0m6.900s	real 0m5.142s	real 0m8.661s	real 0m8.547s	real 0m8.444s
user 0m2.549s	user 0m21.586s	user 0m27.854s	user 0m10.424s	user 0m26.768s	user 0m32.726s
sys 0m1.975s	sys 0m1.565s	sys 0m1.474s	sys 0m1.343s	sys 0m2.236s	sys 0m1.500s
real 0m2.006s	real 0m6.886s	real 0m4.942s	real 0m8.441s	real 0m8.268s	real 0m8.530s
user 0m2.353s	user 0m21.262s	user 0m26.242s	user 0m10.073s	user 0m25.795s	user 0m30.972s
sys 0m1.803s	sys 0m1.539s	sys 0m1.256s	sys 0m1.239s	sys 0m2.152s	sys 0m1.411s
real 0m1.935s	real 0m7.033s	real 0m4.904s	real 0m8.606s	real 0m8.185s	real 0m8.415s
user 0m2.296s	user 0m21.231s	user 0m27.414s	user 0m10.229s	user 0m24.875s	user 0m30.466s
sys 0m1.935s	sys 0m1.702s	sys 0m1.234s	sys 0m1.347s	sys 0m1.968s	sys 0m1.435s

Graceshopper-Elektra			Math_Fluency_App		
Install	QLDB	Query Time	Install	QLDB	Query Time
real 0m11.411s	real 0m7.795s	real 0m5.882s	real 0m2.957s	real 0m7.306s	real 0m10.023s
user 0m12.885s	user 0m25.418s	user 0m30.272s	user 0m3.191s	user 0m22.177s	user 0m33.285s
sys 0m12.589s	sys 0m1.994s	sys 0m1.443s	sys 0m1.946s	sys 0m1.485s	sys 0m1.567s
real 0m10.773s	real 0m7.822s	real 0m5.684s	real 0m2.982s	real 0m7.319s	real 0m10.002s
user 0m11.861s	user 0m24.591s	user 0m28.328s	user 0m3.191s	user 0m21.669s	user 0m31.434s
sys 0m12.600s	sys 0m2.071s	sys 0m1.289s	sys 0m2.592s	sys 0m1.543s	sys 0m1.526s
real 0m11.573s	real 0m7.768s	real 0m5.767s	real 0m2.496s	real 0m7.513s	real 0m10.141s
user 0m13.365s	user 0m23.464s	user 0m26.850s	user 0m2.992s	user 0m23.902s	user 0m31.639s
sys 0m11.613s	sys 0m2.013s	sys 0m1.390s	sys 0m1.381s	sys 0m1.706s	sys 0m1.456s
real 0m11.579s	real 0m7.909s	real 0m5.799s	real 0m2.561s	real 0m7.388s	real 0m9.953s
user 0m13.367s	user 0m23.956s	user 0m28.813s	user 0m3.023s	user 0m21.587s	user 0m32.104s
sys 0m11.507s	sys 0m2.136s	sys 0m1.339s	sys 0m1.814s	sys 0m1.620s	sys 0m1.425s
real 0m11.633s	real 0m7.750s	real 0m5.806s	real 0m2.672s	real 0m7.490s	real 0m10.304s
user 0m13.494s	user 0m25.284s	user 0m29.384s	user 0m3.003s	user 0m23.463s	user 0m32.897s
sys 0m12.138s	sys 0m2.060s	sys 0m1.443s	sys 0m1.736s	sys 0m1.657s	sys 0m1.586s
real 0m10.429s	real 0m7.969s	real 0m5.999s	real 0m2.882s	real 0m7.685s	real 0m10.137s
user 0m12.003s	user 0m25.646s	user 0m30.554s	user 0m3.438s	user 0m22.874s	user 0m30.893s
sys 0m10.399s	sys 0m2.077s	sys 0m1.534s	sys 0m1.634s	sys 0m1.792s	sys 0m1.258s
real 0m11.266s	real 0m7.869s	real 0m5.695s	real 0m2.722s	real 0m7.461s	real 0m10.018s
user 0m12.832s	user 0m25.917s	user 0m29.252s	user 0m3.173s	user 0m23.799s	user 0m33.256s
sys 0m12.057s	sys 0m2.084s	sys 0m1.341s	sys 0m1.564s	sys 0m1.796s	sys 0m1.360s
real 0m10.657s	real 0m7.778s	real 0m5.834s	real 0m2.885s	real 0m7.331s	real 0m10.116s
user 0m12.307s	user 0m24.296s	user 0m30.690s	user 0m3.462s	user 0m23.027s	user 0m33.333s
sys 0m10.609s	sys 0m1.913s	sys 0m1.641s	sys 0m1.798s	sys 0m1.672s	sys 0m1.503s
real 0m11.379s	real 0m7.914s	real 0m5.749s	real 0m2.570s	real 0m7.293s	real 0m9.691s
user 0m13.284s	user 0m23.948s	user 0m27.625s	user 0m2.995s	user 0m22.935s	user 0m29.432s
sys 0m11.607s	sys 0m1.960s	sys 0m1.332s	sys 0m1.440s	sys 0m1.646s	sys 0m1.220s
real 0m10.142s	real 0m7.932s	real 0m5.708s	real 0m2.444s	real 0m7.536s	real 0m10.235s
user 0m11.604s	user 0m23.992s	user 0m27.512s	user 0m2.895s	user 0m22.043s	user 0m33.485s
sys 0m10.757s	sys 0m2.065s	sys 0m1.261s	sys 0m1.472s	sys 0m1.722s	sys 0m1.579s

NetSteam			property-manage		
Install	QLDB	Query Time	Install	QLDB	Query Time
real 0m11.003s user 0m13.064s sys 0m1.606s	real 0m8.410s user 0m28.245s sys 0m2.308s	real 0m6.155s user 0m31.081s sys 0m1.381s	real 0m10.658s user 0m12.716s sys 0m1.601s	real 0m9.149s user 0m28.052s sys 0m2.514s	real 0m7.442s user 0m30.424s sys 0m1.436s
real 0m11.016s user 0m13.240s sys 0m1.608s	real 0m8.498s user 0m27.928s sys 0m2.396s	real 0m6.264s user 0m30.930s sys 0m1.598s	real 0m11.130s user 0m13.114s sys 0m1.860s	real 0m8.931s user 0m29.058s sys 0m2.491s	real 0m7.641s user 0m31.780s sys 0m1.436s
real 0m10.628s user 0m12.755s sys 0m1.506s	real 0m8.410s user 0m25.213s sys 0m2.230s	real 0m6.272s user 0m29.528s sys 0m1.540s	real 0m10.674s user 0m12.862s sys 0m1.517s	real 0m9.063s user 0m29.383s sys 0m2.589s	real 0m7.509s user 0m32.237s sys 0m1.362s
real 0m10.498s user 0m12.587s sys 0m1.495s	real 0m8.450s user 0m27.427s sys 0m2.233s	real 0m6.348s user 0m31.182s sys 0m1.459s	real 0m22.907s user 0m14.907s sys 0m1.723s	real 0m8.859s user 0m28.432s sys 0m2.284s	real 0m7.593s user 0m31.413s sys 0m1.287s
real 0m10.942s user 0m13.029s sys 0m1.720s	real 0m8.563s user 0m26.890s sys 0m2.419s	real 0m6.382s user 0m29.999s sys 0m1.474s	real 0m11.036s user 0m13.131s sys 0m1.638s	real 0m8.823s user 0m28.182s sys 0m2.348s	real 0m7.828s user 0m31.304s sys 0m1.446s
real 0m11.067s user 0m13.432s sys 0m1.496s	real 0m8.513s user 0m27.484s sys 0m2.418s	real 0m6.208s user 0m29.898s sys 0m1.286s	real 0m10.890s user 0m13.076s sys 0m1.713s	real 0m8.997s user 0m29.630s sys 0m2.353s	real 0m7.706s user 0m33.316s sys 0m1.684s
real 0m10.742s user 0m12.947s sys 0m1.603s	real 0m8.466s user 0m26.769s sys 0m2.337s	real 0m6.080s user 0m29.782s sys 0m1.266s	real 0m10.098s user 0m11.846s sys 0m1.487s	real 0m9.020s user 0m28.422s sys 0m2.522s	real 0m7.733s user 0m32.422s sys 0m1.533s
real 0m9.926s user 0m11.754s sys 0m1.476s	real 0m8.154s user 0m25.842s sys 0m2.053s	real 0m6.289s user 0m30.434s sys 0m1.601s	real 0m10.691s user 0m12.670s sys 0m1.698s	real 0m8.966s user 0m27.840s sys 0m2.360s	real 0m7.779s user 0m31.788s sys 0m1.596s
real 0m13.321s user 0m13.650s sys 0m1.479s	real 0m8.377s user 0m26.024s sys 0m2.276s	real 0m6.123s user 0m30.409s sys 0m1.418s	real 0m11.067s user 0m12.838s sys 0m1.677s	real 0m9.163s user 0m26.896s sys 0m2.610s	real 0m7.666s user 0m30.960s sys 0m1.445s
real 0m11.349s user 0m13.210s sys 0m1.340s	real 0m8.428s user 0m25.486s sys 0m2.257s	real 0m6.162s user 0m30.325s sys 0m1.343s	real 0m11.313s user 0m13.144s sys 0m1.605s	real 0m8.788s user 0m28.734s sys 0m2.368s	real 0m7.621s user 0m31.450s sys 0m1.398s

wall			youtubeclone		
Install	QLDB	Query Time	Install	QLDB	Query Time
real 0m16.770s	real 0m7.658s	real 0m5.719s	real 0m2.892s	real 0m7.108s	real 0m6.191s
user 0m9.730s	user 0m24.902s	user 0m29.430s	user 0m3.502s	user 0m24.347s	user 0m27.984s
sys 0m6.870s	sys 0m2.019s	sys 0m1.422s	sys 0m1.905s	sys 0m1.767s	sys 0m1.341s
real 0m17.404s	real 0m7.752s	real 0m5.920s	real 0m3.186s	real 0m7.014s	real 0m6.056s
user 0m10.657s	user 0m23.775s	user 0m29.759s	user 0m3.882s	user 0m23.523s	user 0m28.593s
sys 0m6.709s	sys 0m2.007s	sys 0m1.495s	sys 0m2.247s	sys 0m1.700s	sys 0m1.400s
real 0m16.907s	real 0m7.583s	real 0m5.742s	real 0m2.926s	real 0m6.987s	real 0m6.402s
user 0m9.638s	user 0m24.011s	user 0m28.888s	user 0m3.613s	user 0m22.635s	user 0m28.533s
sys 0m6.544s	sys 0m2.022s	sys 0m1.300s	sys 0m1.843s	sys 0m1.656s	sys 0m1.395s
real 0m19.291s	real 0m7.559s	real 0m5.864s	real 0m2.874s	real 0m7.122s	real 0m6.315s
user 0m10.778s	user 0m24.830s	user 0m28.538s	user 0m3.693s	user 0m23.734s	user 0m27.837s
sys 0m7.192s	sys 0m2.017s	sys 0m1.350s	sys 0m1.600s	sys 0m1.824s	sys 0m1.239s
real 0m18.052s	real 0m7.422s	real 0m5.684s	real 0m2.680s	real 0m6.829s	real 0m6.273s
user 0m10.614s	user 0m24.567s	user 0m28.092s	user 0m3.393s	user 0m23.008s	user 0m28.773s
sys 0m7.106s	sys 0m1.973s	sys 0m1.313s	sys 0m1.429s	sys 0m1.517s	sys 0m1.629s
real 0m18.024s	real 0m7.688s	real 0m5.906s	real 0m3.064s	real 0m7.082s	real 0m6.312s
user 0m10.881s	user 0m24.239s	user 0m28.830s	user 0m3.418s	user 0m23.157s	user 0m29.004s
sys 0m7.557s	sys 0m2.043s	sys 0m1.531s	sys 0m2.480s	sys 0m1.687s	sys 0m1.357s
real 0m17.687s	real 0m7.649s	real 0m5.940s	real 0m2.836s	real 0m7.102s	real 0m6.490s
user 0m10.695s	user 0m24.003s	user 0m28.382s	user 0m3.568s	user 0m23.123s	user 0m29.695s
sys 0m7.459s	sys 0m2.166s	sys 0m1.387s	sys 0m1.758s	sys 0m1.703s	sys 0m1.652s
real 0m17.266s	real 0m7.558s	real 0m5.823s	real 0m2.855s	real 0m7.217s	real 0m6.360s
user 0m10.170s	user 0m24.295s	user 0m28.224s	user 0m3.456s	user 0m23.559s	user 0m30.007s
sys 0m6.561s	sys 0m1.947s	sys 0m1.283s	sys 0m1.972s	sys 0m1.852s	sys 0m1.592s
real 0m17.494s	real 0m7.568s	real 0m5.792s	real 0m2.821s	real 0m6.914s	real 0m6.279s
user 0m10.752s	user 0m24.158s	user 0m28.084s	user 0m3.424s	user 0m22.914s	user 0m28.492s
sys 0m6.487s	sys 0m1.978s	sys 0m1.398s	sys 0m1.861s	sys 0m1.652s	sys 0m1.423s
real 0m16.911s	real 0m7.745s	real 0m5.842s	real 0m2.820s	real 0m6.989s	real 0m6.210s
user 0m9.803s	user 0m24.355s	user 0m26.820s	user 0m3.546s	user 0m22.581s	user 0m27.702s
sys 0m6.659s	sys 0m2.174s	sys 0m1.249s	sys 0m1.567s	sys 0m1.672s	sys 0m1.318s

RQ4: How much do the refactorings affect page load times?

First, we show the raw data observations yielding the averages reported in the chapter. Times were estimated from the screenshot timeline described in Figure S1. We drew observations 10 times to the nearest quarter second.

In the following pages you'll find screenshots of the front-end for each of the four applications considered in the **RQ4** case study. Screenshots are of the Chrome Developer Tools "Performance" tab; there are before and after screenshots at each database scale. Figure S1 explains the views in more detail, with an example from **youtubeclone** at the 100 scale.

		Scale = 10 (Before)		Scale = 10 (After)		Scale = 100 (Before)		Scale = 100 (After)		Scale = 1000 (Before)		Scale = 1000 (After)	
Application	Link to Fn Under Test	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev
eventbright	getEvents4	0.375	0.1317615692	0.25	0	1	0	0.25	0	7.725	0.9010025281	1.375	0.1317615692
		0.25		0.25		1		0.25		7.5		1.5	
		0.5		0.25		1		0.25		7.75		1.25	
		0.25		0.25		1		0.25		7.5		1.25	
		0.25		0.25		1		0.25		7.5		1.25	
		0.25		0.25		1		0.25		7.25		1.25	
		0.5		0.25		1		0.25		7.5		1.5	
		0.5		0.25		1		0.25		7.25		1.5	
		0.25		0.25		1		0.25		10.25		1.5	
		0.5		0.25		1		0.25		7.25		1.5	
		0.5		0.25		1		0.25		7.5		1.25	
property-manage	getProperties	0.25	0	0.25	0	0.5	0	0.5	0	2.95	0.1972026594	2.825	0.1207614729
		0.25		0.25		0.5		0.5		2.75		2.75	
The times were very consisten in this application, likely because the N+1 pattern query (i.e., the query in the loop) was not used to construct the response.		0.25		0.25		0.5		0.5		3		3	
		0.25		0.25		0.5		0.5		3		2.75	
		0.25		0.25		0.5		0.5		2.75		3	
		0.25		0.25		0.5		0.5		3.25		2.75	
		0.25		0.25		0.5		0.5		3.25		2.75	
		0.25		0.25		0.5		0.5		3		2.75	
		0.25		0.25		0.5		0.5		2.75		2.75	
		0.25		0.25		0.5		0.5		2.75		2.75	
		0.25		0.25		0.5		0.5		3		3	
NetSteam	getReviewsForVid									3.825	0.2058181506	2	0.2041241452
		*		*		0.5		*		4		2.25	
*** indicates that the reviews loaded before the animation completed**		*		*		*		*		3.75		2.25	
		*		*		*		*		3.75		1.75	
		*		*		*		*		4.25		2	
		*		*		*		*		3.75		1.75	
		*		*		*		*		3.75		2	
		*		*		*		*		3.5		2	
		*		*		0.5		*		3.75		2.25	
		*		*		*		*		3.75		1.75	
		*		*		*		*		4		2	
youtube-clone	searchUser	1.175	0.1207614729	0.525	0.141911553	3.825	0.1687371394	0.825	0.2371708245	19.875	0.242956329	1.95	0.2838231061
		1		0.5		3.75		0.5		19.75		1.25	
		1.25		0.5		4		1		19.5		2	
		1.25		0.75		3.75		1		19.75		2	
		1.25		0.5		3.75		0.75		20.25		2	
		1.25		0.5		4		1		20		2.25	
		1.25		0.5		3.75		0.5		19.75		2	
		1		0.5		4		0.5		19.75		1.75	
		1		0.25		4		1		20.25		2	
		1.25		0.75		3.75		1		20		2.25	
		1.25		0.5		3.5		1		19.75		2	

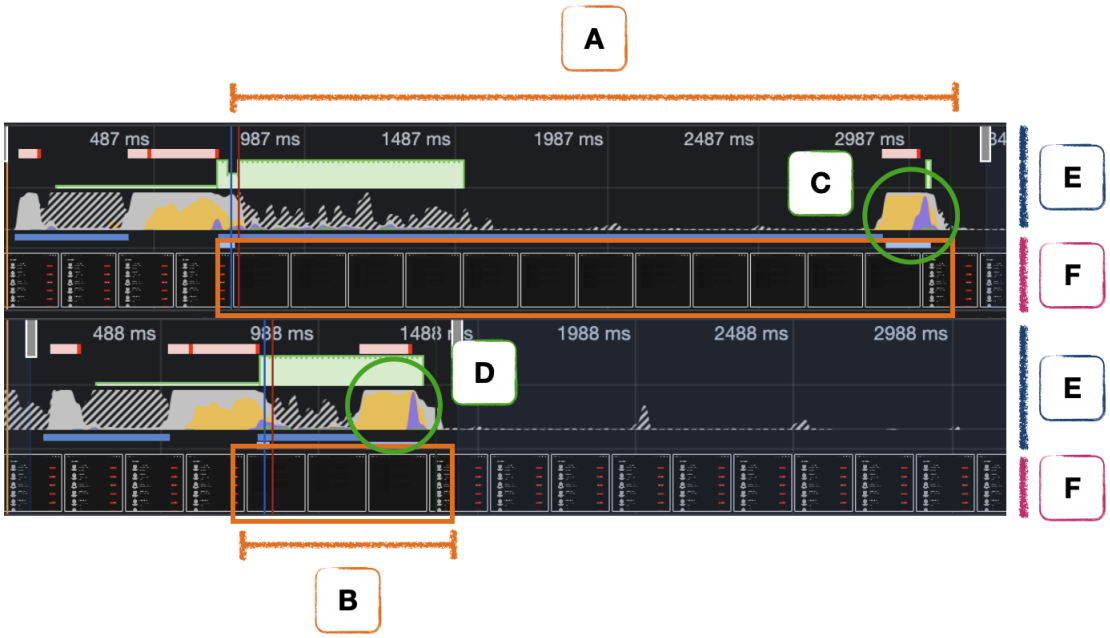
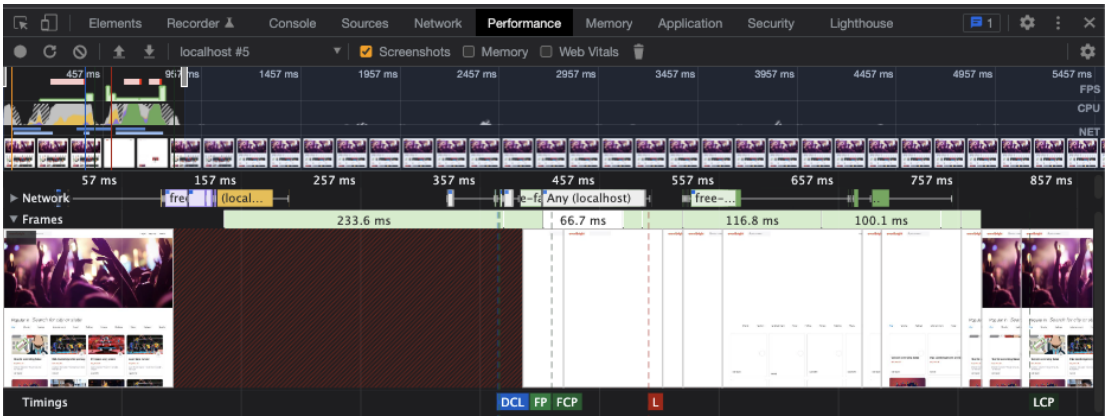
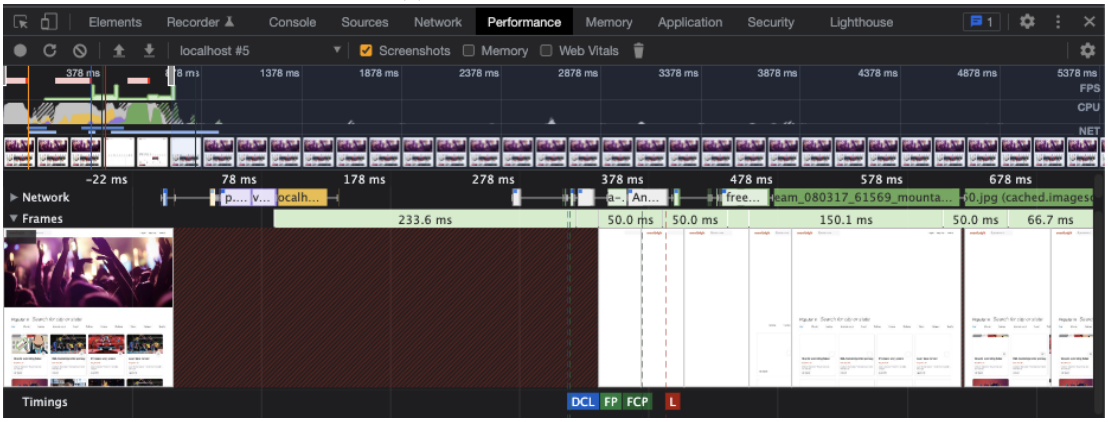


Figure S1: Two screenshots from the Chrome DevTools' Performance Tab profiling a search turning up 100 users in **youtubeclone**. The profile corresponding to the original code is on top, and the refactored one is on the bottom. The two (E) labels show time series of application activity, where higher values correspond to more CPU cycles. (C) and (D) show spikes in activity when the the HTTP response was received by the client before and after refactoring, resp. The two (F) labels show a series of screenshots taken of the front-end as it loads and is populated by data. (A) and (B) show the span of time that the screen was idle before and after refactoring, resp, and the two boxes in the timelines highlights that the screen is empty during that span.

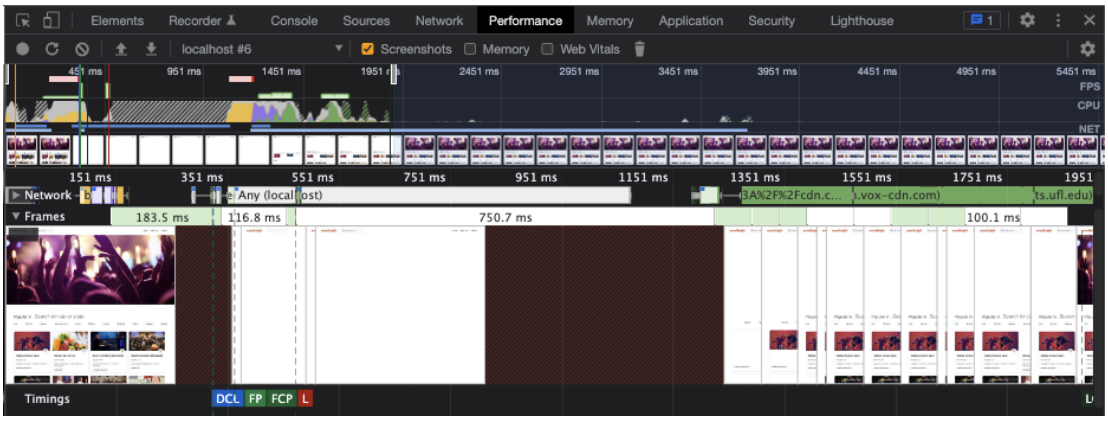


(a) eventbright, 10 scale, before

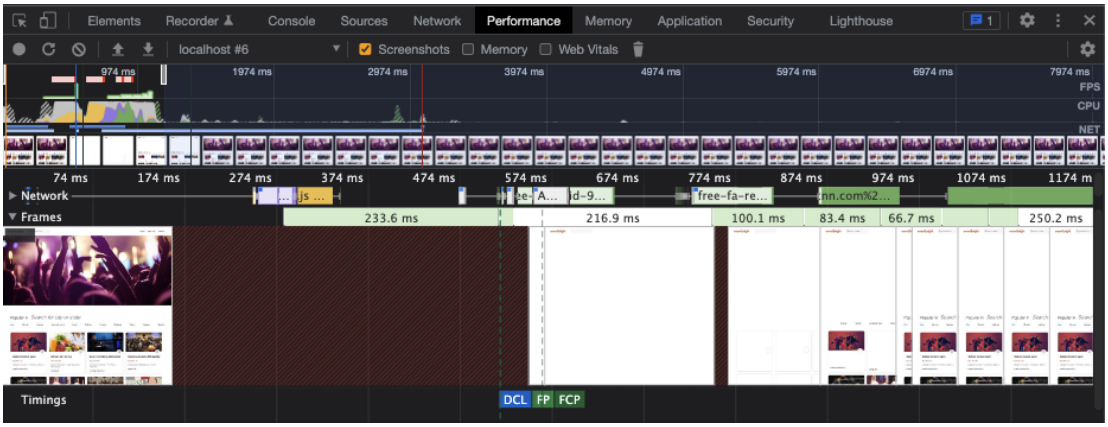


(b) eventbright, 10 scale, after

Figure S2: The front-end load time difference is imperceptible here.

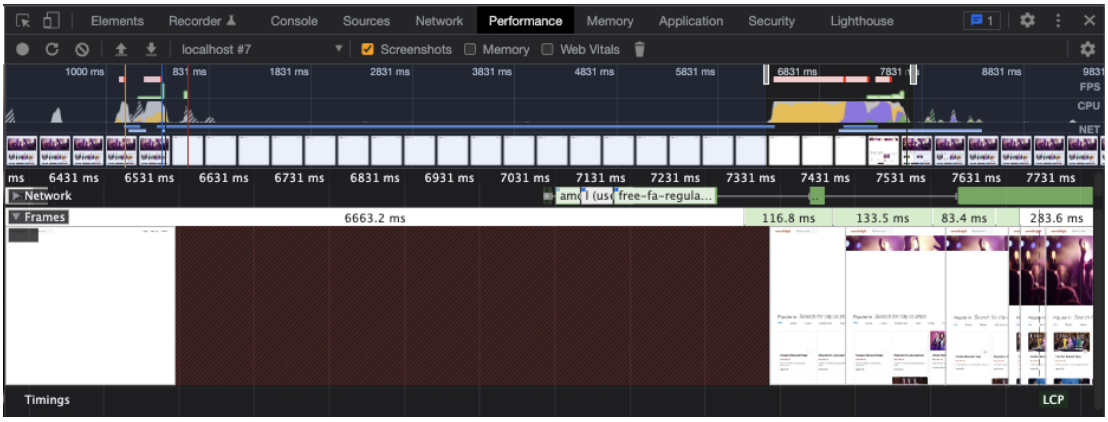


(a) eventbright, 100 scale, before

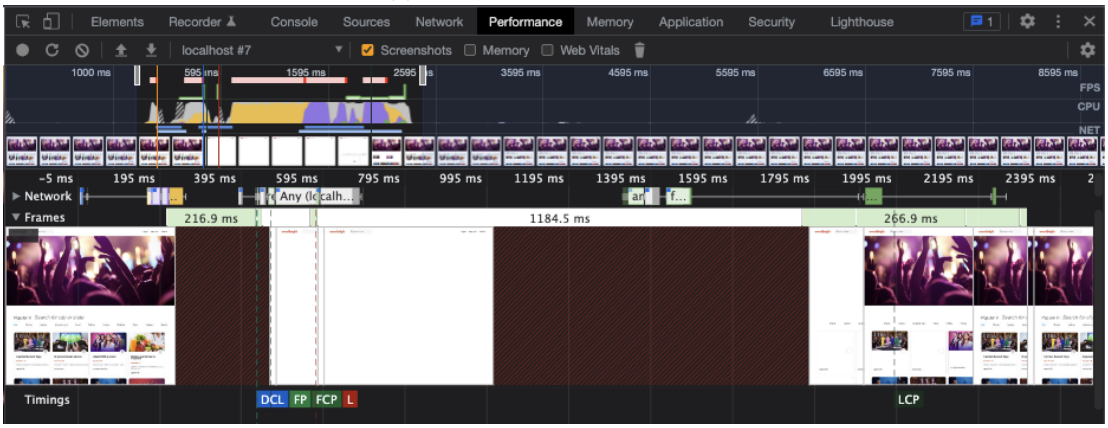


(b) eventbright, 100 scale, after

Figure S3: Here, the timeline clearly shows that the number of idle frames is quite different before and after refactoring. The page appears to become populated ∼0.8s faster after refactoring.

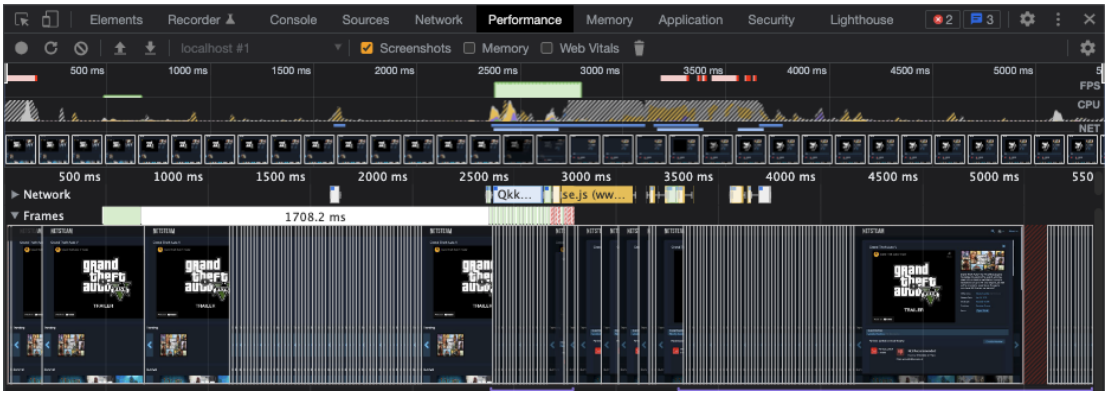


(a) eventbright, 1000 scale, before



(b) eventbright, 1000 scale, after

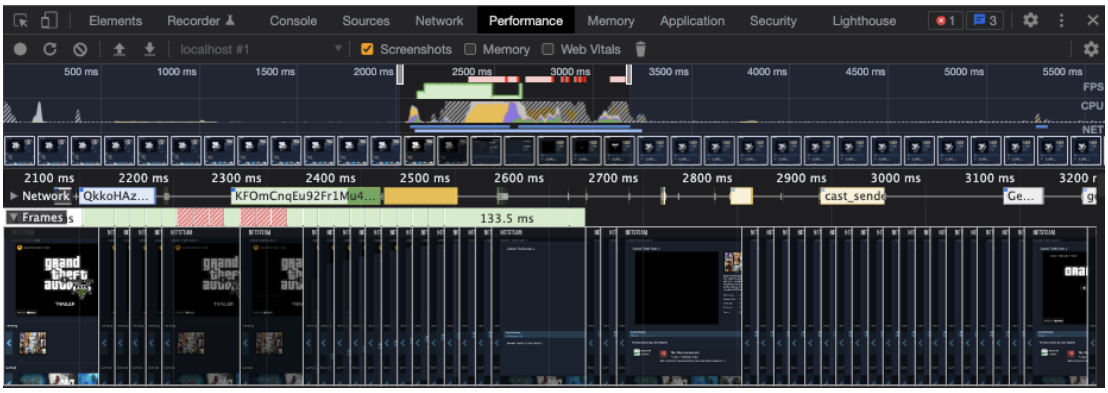
Figure S4: There is a huge difference in page load times in this configuration. Each of these timelines spans approximately 10s, and the time taken before is noticeably longer before refactoring, as the refactored page appears to load 4 to 5 times faster.



(a) NetSteam, 10 scale, before

(b) NetSteam, 10 scale, after

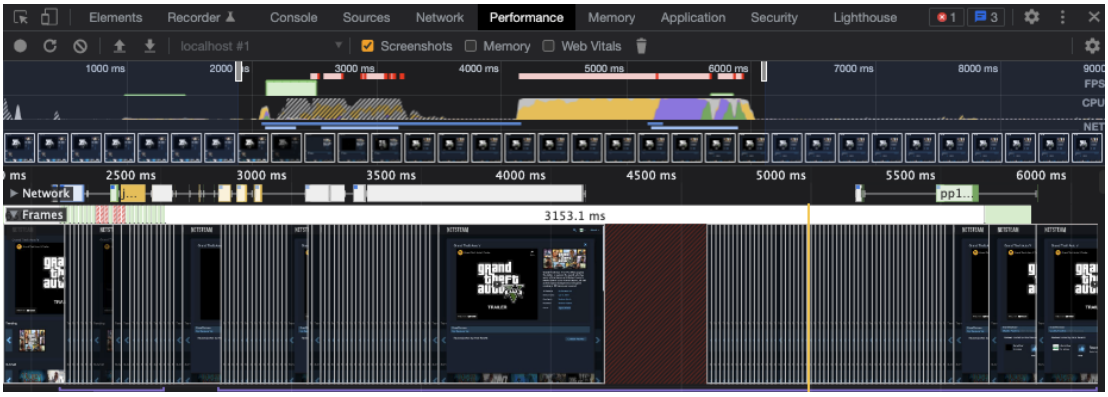
Figure S5: Judging by the time taken and the activity profiles, there is no measurable difference in front-end load times at this scale.



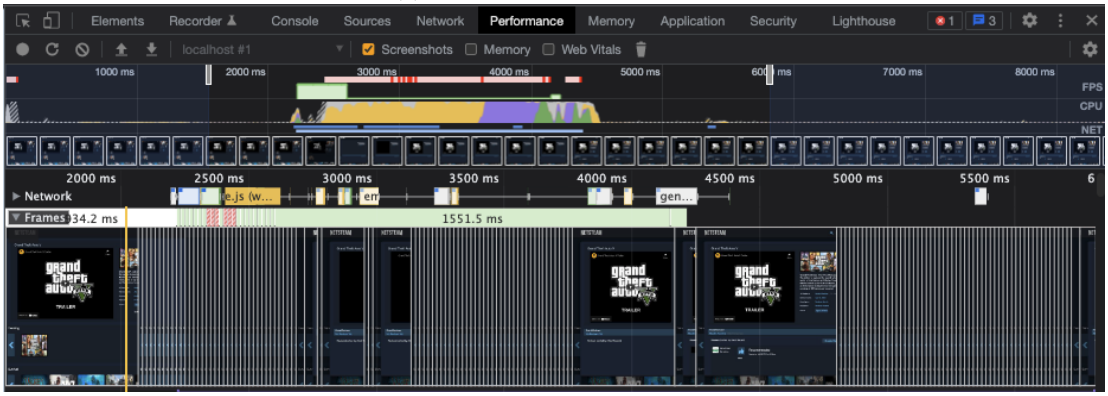
(a) NetSteam, 100 scale, before

(b) NetSteam, 100 scale, after

Figure S6: Here, there is no measurable difference in when the page is populated with data. There is an animation, which you can see in the screenshot timeline, with many screenshots that are slightly different; this animation essentially hides any improvement to page loading. You can see by the lack of dead time in the activity timeline that the data was received from the server more quickly after refactoring.

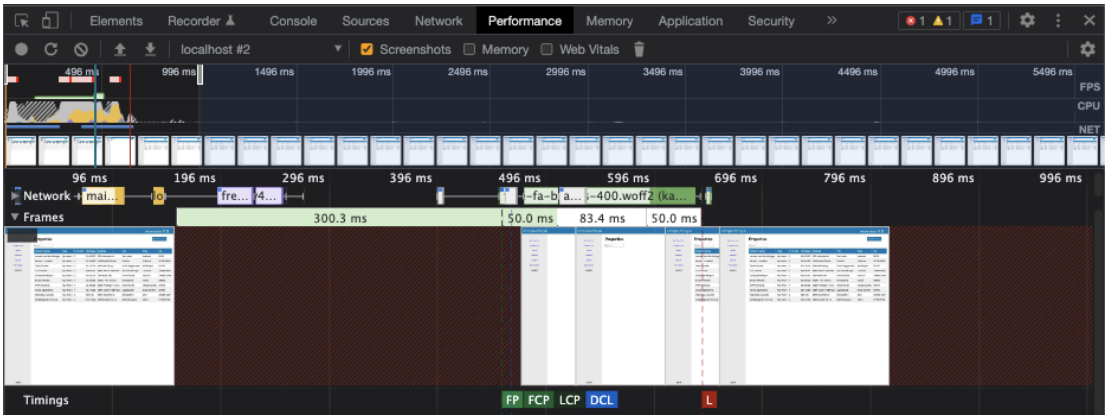


(a) NetSteam, 1000 scale, before

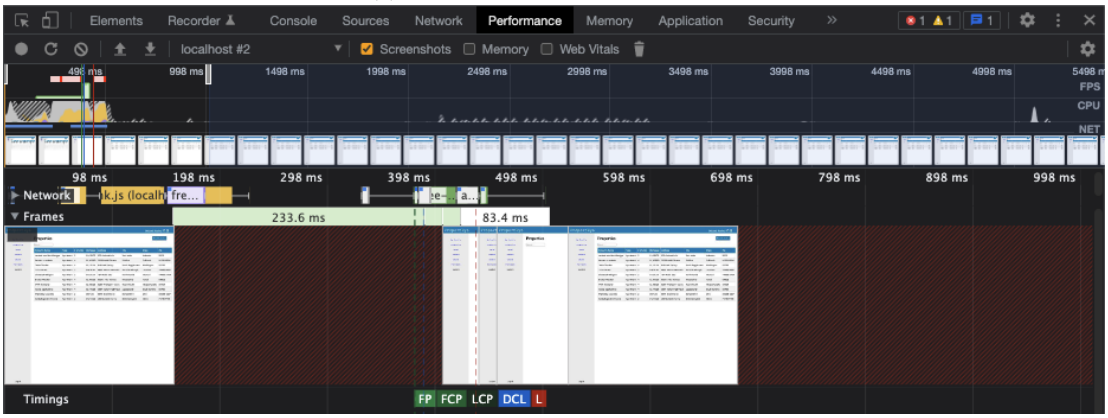


(b) NetSteam, 1000 scale, after

Figure S7: There is a notable difference in page load time here, of approximately 2s. To see it, note the activity time series: at around the 2s mark, the HTTP request to load reviews was sent. In the "before" case, it takes much longer for the server to respond, before a flurry of activity seen on the time series in yellow. In the "after" case, in contrast, the server almost immediately responds, allowing the page to build the UI more quickly.

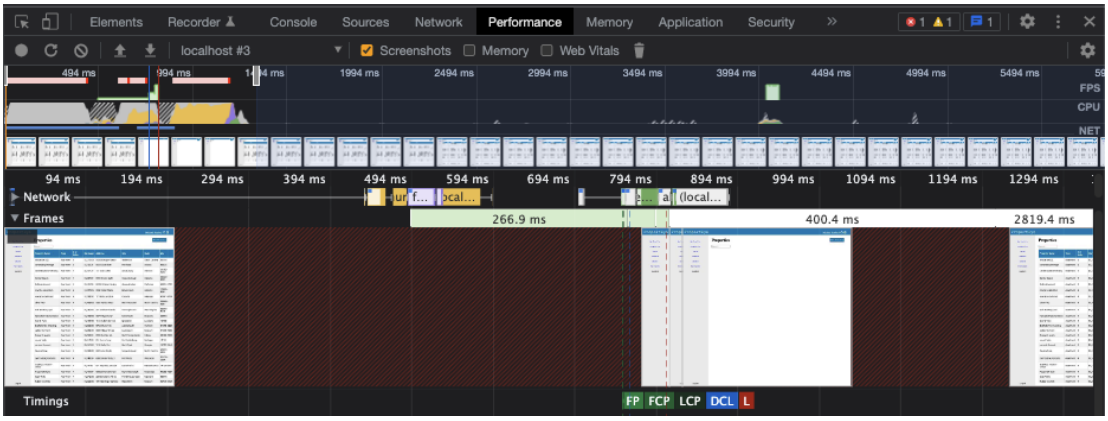


(a) property-manage, 10 scale, before

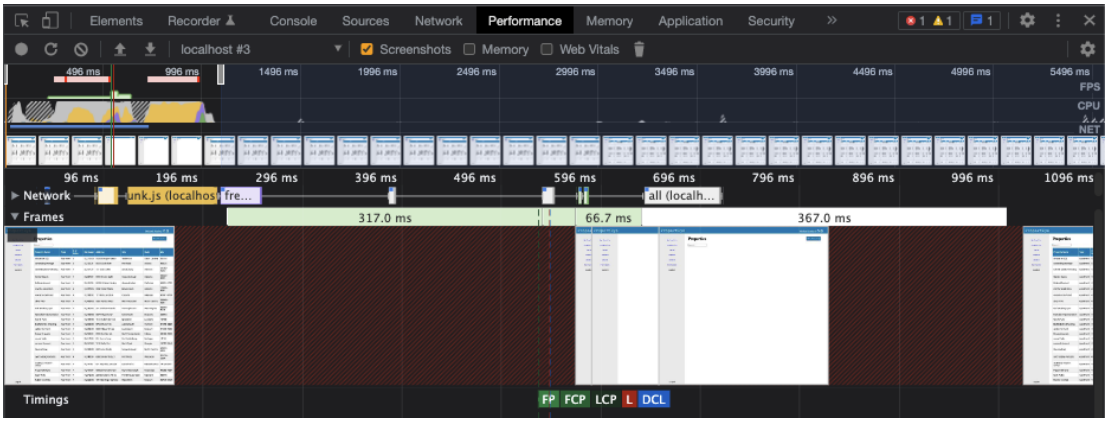


(b) property-manage, 10 scale, after

Figure S8: The difference in load times is just noise here.

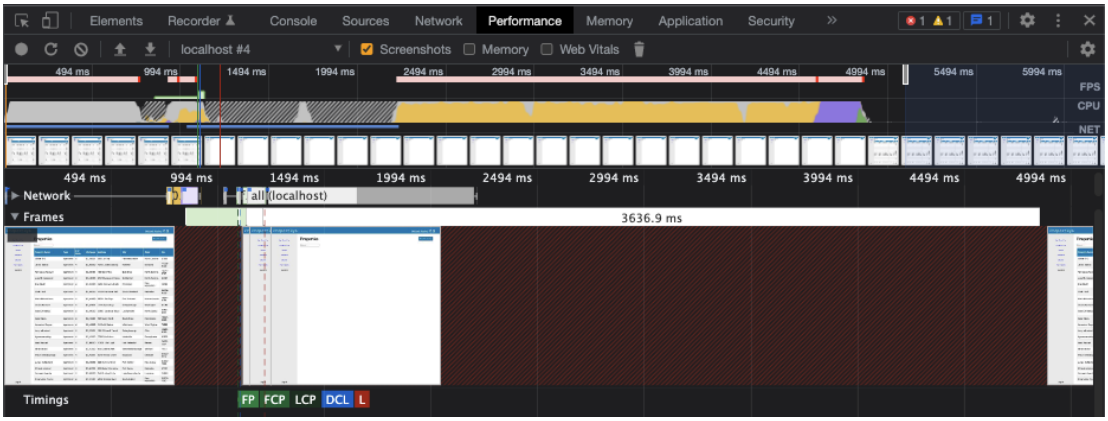


(a) property-manage, 100 scale, before

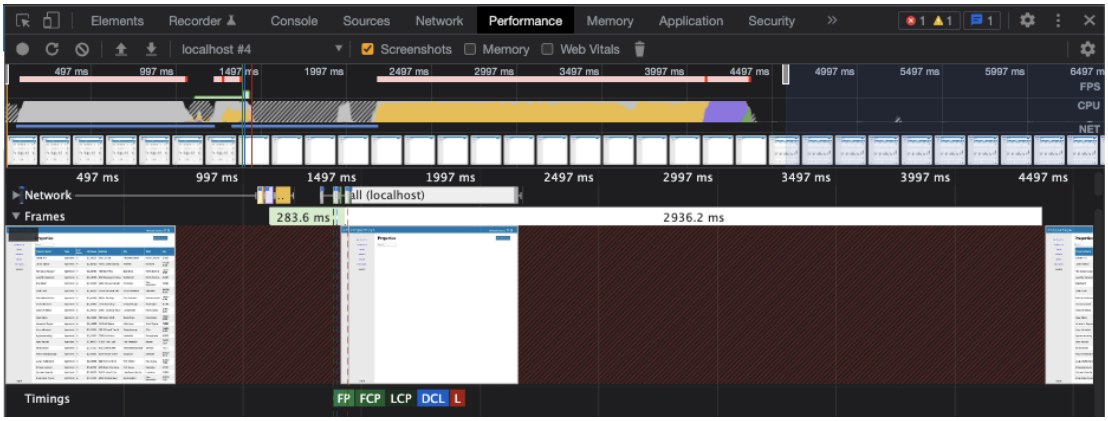


(b) property-manage, 100 scale, after

Figure S9: The difference in load times is just noise here.

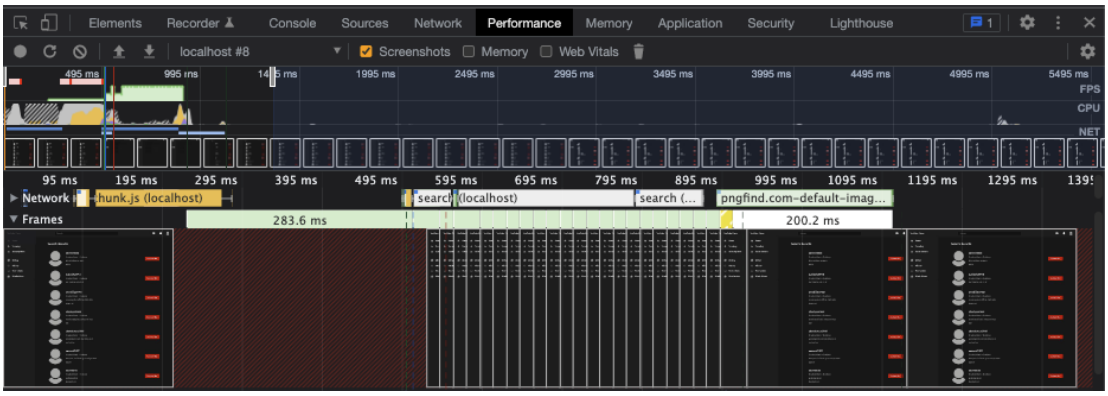


(a) property-manage, 1000 scale, before

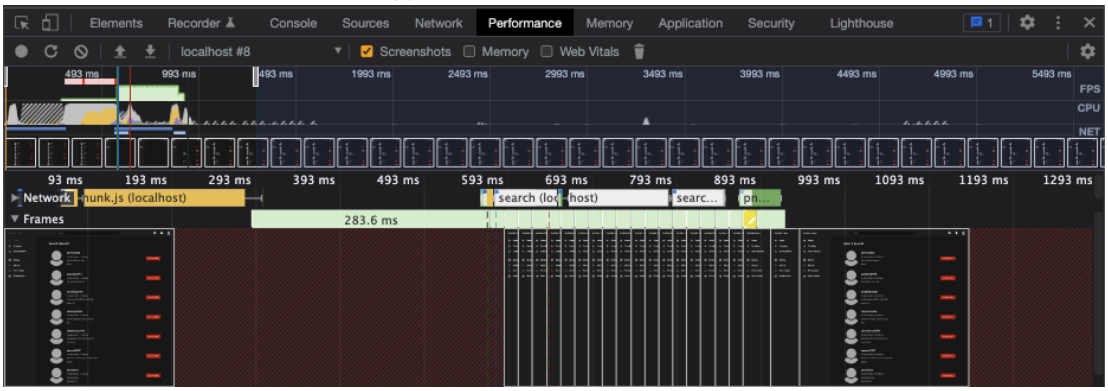


(b) property-manage, 1000 scale, after

Figure S10: The difference in load times is just noise here.

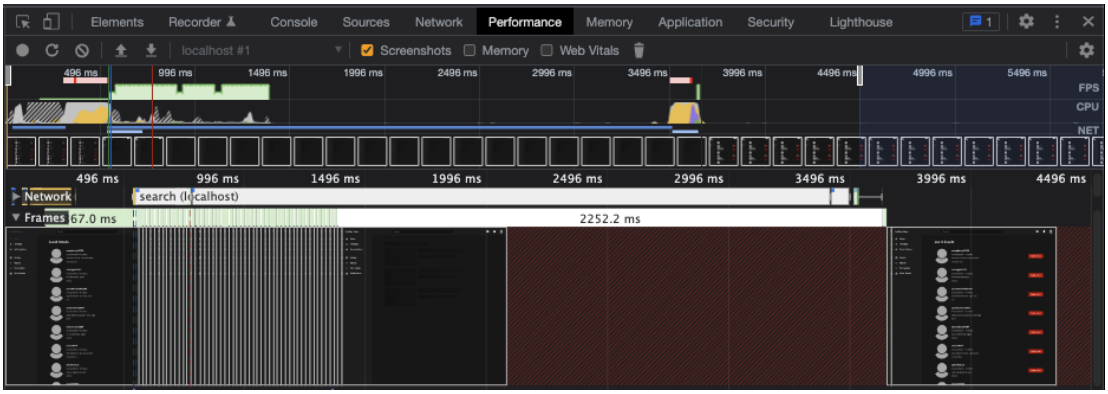


(a) youtubeclone, 10 scale, before

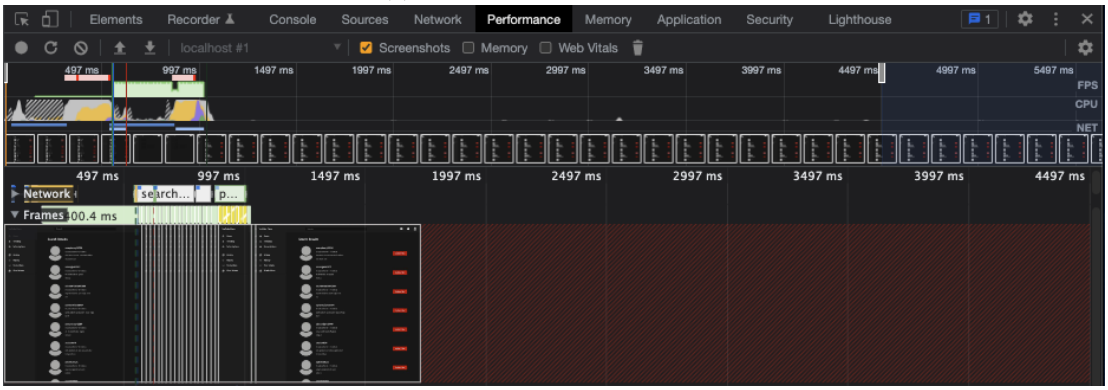


(b) youtubeclone, 10 scale, after

Figure S11: There isn't a notable difference in load times here. The server responds a little faster after refactoring, but it would likely not be noticeable to users.

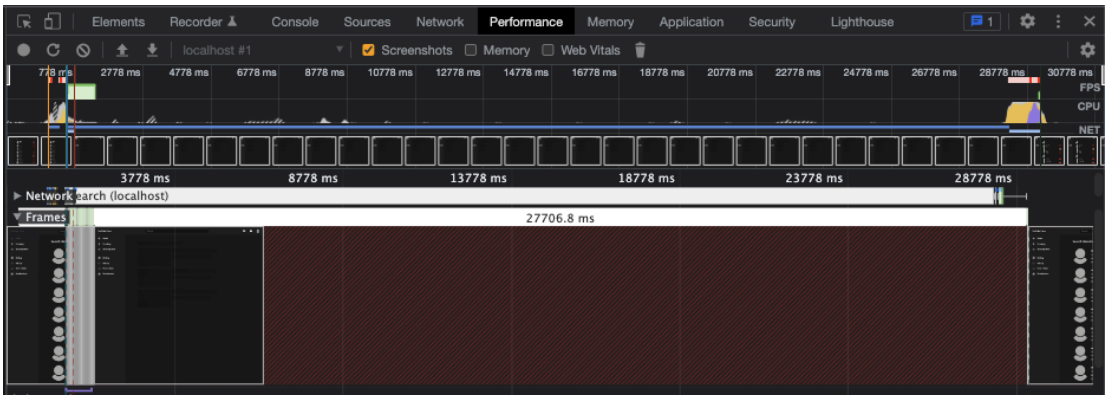


(a) youtubeclone, 100 scale, before

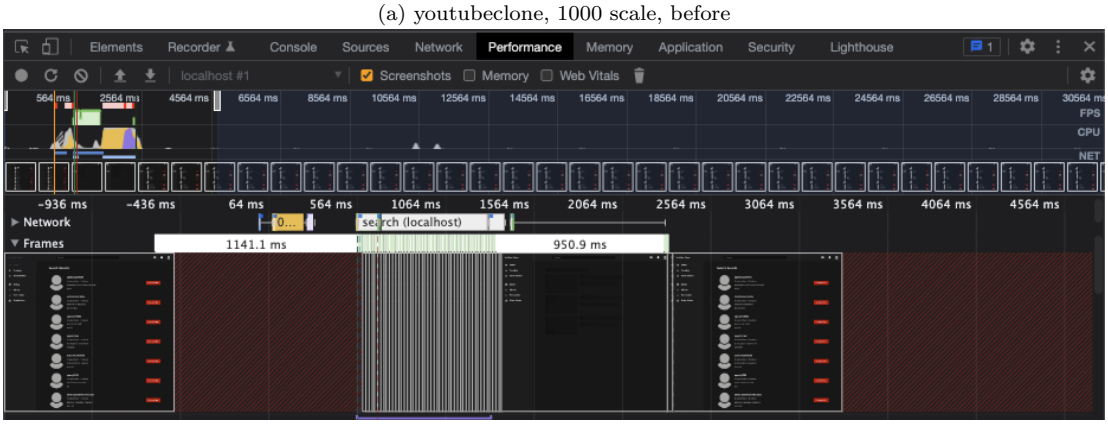


(b) youtubeclone, 100 scale, after

Figure S12: Here we see a large difference in load times. Before refactoring, there is a lot of dead time while the page waits for the server to respond which can be seen in the activity graph, and also the number of idle frames. The page appears to load 2.5s faster with the refactored code.



(a) youtubeclone, 1000 scale, before



(b) youtubeclone, 1000 scale, after

Figure S13: The load time difference in this configuration is extreme, the page loads nearly 10 times faster in the refactored version.

Appendix C

Software Debloating

This is the supporting material for Chapter 6. It includes the following content:

- Results for clients of all projects;
- Results for clients of all projects with guarded execution mode enabled;
- Example of the dynamically loaded code when running in guarded execution mode.

The artifact contains the full source code of of our tool, the full data from our experiments and our associated data processing code, and an environment set up to easily rerun our experiments and test the tool. A link can be found in the chapter.

Results for clients of all projects

	Client		Static Callgraph					Dynamic Callgraph				
Proj	**Client Proj**	**T(s)**	**T(s)**	**SD**	**Fls**	**Fcts**	**Exp.KB**	**T(s)**	**SD**	**Fls**	**Fcts**	**Exp.KB**
memfs	artgen	9.78	10.28	5%	0	0	0.00	10.26	5%	0	0	0.00
	class-component-converter	9.73	10.45	7%	15	0	118.96	9.82	1%	15	0	118.96
	jaxon-ts	19.23	19.29	0%	15	0	118.96	20.70	7%	15	0	118.96
	webpack-dev-middleware	11.86	12.71	7%	15	0	118.96	12.75	7%	15	0	118.96
	zenobia-ts	18.63	18.97	2%	15	0	118.96	21.19	12%	15	0	118.96
fs-nextra	adset-DEPRECATED	0.34	0.36	5%	8	0	13.11	0.33	-2%	0	0	0.00
	AdvancedJS	0.29	0.29	0%	0	0	0.00	0.31	7%	0	0	0.00
	better-klasa	0.27	0.34	22%	0	0	0.00	0.31	14%	0	0	0.00
	core	1.53	1.86	18%	8	0	13.11	1.54	1%	0	0	0.00
	klasa	2.15	2.15	0%	8	0	13.11	2.16	1%	0	0	0.00
body-parser	appium-base-driver	8.66	10.04	14%	39	0	146.10	9.26	6%	8	0	6.69
	express	1.05	1.89	45%	48	0	231.69	1.21	14%	14	0	79.70
	karma	2.08	2.12	2%	40	0	199.57	2.09	1%	12	0	79.03
	moleculer-web	5.80	6.46	10%	48	0	231.69	6.38	9%	14	0	79.70
	typescript-rest	13.17	14.89	12%	48	0	231.69	14.48	9%	14	0	79.70
comm ander	html-minifier	9.13	10.20	11%	0	0	0.00	10.13	10%	0	0	0.00
	lint-staged	20.23	20.30	0%	0	0	0.00	20.56	2%	0	0	0.00
	metalsmith	1.44	1.47	2%	0	0	0.00	1.47	2%	0	0	0.00
	nunjucks	35.25	35.17	0%	0	0	0.00	35.41	0%	0	0	0.00
	sheetjs	23.15	23.31	1%	0	0	0.00	23.35	1%	0	0	0.00
memory-fs	astroturf	19.94	20.11	1%	16	5	45.92	23.88	17%	9	0	16.86
	mochapack	52.49	52.47	0%	16	2	43.82	52.70	0%	9	0	16.86
	rax	23.35	25.09	7%	0	0	0.00	24.37	4%	0	0	0.00
	vue-builder	1.98	1.98	0%	16	6	46.85	2.02	2%	9	0	16.86
	webpack	340.46	342.72	1%	0	0	0.00	342.96	1%	0	0	0.00
glob	copyfiles	1.18	1.45	18%	6	10	18.44	1.39	15%	2	0	8.68
	dot-object	2.02	2.12	5%	6	9	11.85	1.89	-7%	2	0	8.68
	node-glob-all	0.28	0.29	4%	6	7	8.87	0.27	-2%	2	0	8.68
	replace-in-file	2.27	2.52	10%	6	10	17.04	2.29	1%	2	0	8.68
	stylus	2.67	3.07	13%	6	1	14.10	2.69	1%	2	0	8.68
redux	Choices	5.06	5.16	2%	1	0	20.06	5.05	0%	1	0	20.06
	found	30.61	31.83	4%	1	0	20.06	31.34	2%	1	0	20.06
	Griddle	8.93	8.91	0%	1	0	20.06	9.03	1%	1	0	20.06
	react-beautiful-dnd	61.70	63.49	3%	2	0	20.06	62.12	1%	2	0	20.06
	redux-ignore	0.57	0.58	2%	1	0	20.06	0.59	3%	1	0	20.06
css-loader	astroturf	18.32	20.38	10%	88	0	336.11	20.23	9%	215	0	787.53
	custom-react-script	1.65	1.72	4%	0	0	0.00	1.71	3%	0	0	0.00
	docusaurus	135.03	134.64	0%	0	0	0.00	135.55	0%	0	0	0.00
	playroom	2.65	2.63	-1%	0	0	0.00	2.64	0%	0	0	0.00
	powerbi-visual-tools	442.16	441.07	0%	0	0	0.00	449.38	2%	0	0	0.00
	decompress-zip	0.70	0.74	6%	1	0	63.25	0.78	10%	0	5	2.98
	downshift	1.43	1.44	1%	1	0	63.25	1.44	1%	0	1	0.88

q	node-ping	3.80	4.20	10%	1	0	63.25	4.08	7%	0	6	2.86
	passport-saml	0.41	0.44	6%	0	0	0.00	0.42	2%	0	0	0.00
	requestify	2.92	2.99	2%	1	0	63.25	3.05	4%	0	2	0.86
send	connect-gzip-static	0.51	0.64	20%	18	10	27.97	0.53	2%	4	0	3.71
	gitbook	4.91	4.92	0%	0	0	0.00	4.91	0%	0	0	0.00
	lasso	13.26	13.39	1%	0	0	0.00	13.19	-1%	0	0	0.00
	node-restify	23.24	24.71	6%	18	11	28.89	24.29	4%	4	0	3.71
	serve-static	0.62	0.64	3%	18	18	33.07	0.64	2%	4	0	3.71
serve-favicon	appium-base-driver	8.34	8.39	1%	2	0	3.11	8.38	1%	0	0	0.00
	enb	3.87	3.91	1%	2	0	0.00	3.90	1%	0	0	0.00
	express-octoblu	0.81	0.85	6%	2	3	0.00	0.82	2%	0	0	0.00
	loopback	29.45	29.64	1%	2	0	3.11	29.50	0%	0	0	0.00
	server	9.91	10.08	2%	2	3	3.11	10.02	1%	0	0	0.00
morgan	appium-base-driver	8.58	8.58	0%	10	0	16.34	8.54	0%	2	0	3.04
	ember-cli	89.96	92.96	3%	10	5	19.55	93.62	4%	2	0	3.04
	gulp-contrib-connect	0.89	0.92	3%	10	0	16.34	0.94	6%	2	0	3.04
	json-server	19.79	20.40	3%	10	1	17.32	19.92	1%	2	0	3.04
	superstatic	0.94	0.97	3%	10	1	16.83	1.04	10%	2	0	3.04
serve-static	connect-gzip-static	0.56	0.57	3%	23	1	45.53	0.57	3%	4	0	3.71
	gulp-connect	0.69	0.70	2%	23	1	45.53	0.70	2%	4	0	3.71
	moleculer-web	6.37	6.44	1%	23	1	45.53	6.39	0%	4	0	3.71
	reload	8.00	8.06	1%	23	1	45.53	8.16	2%	4	0	3.71
	soap	1.42	1.45	2%	0	0	0.00	1.44	1%	0	0	0.00
prop-types	react-dates	18.61	18.67	0%	7	0	37.25	18.58	0%	7	0	37.25
	react-loadable	52.16	51.58	-1%	7	0	37.25	52.73	1%	7	0	37.25
	react-redux	7.02	7.08	1%	7	0	37.25	7.12	1%	7	0	37.25
	redux-form	15.48	15.24	-2%	7	0	37.25	15.93	3%	7	0	37.25
	wd	18.06	20.45	12%	7	0	37.25	18.53	3%	7	0	37.25
compression	cordova-serve	0.62	0.62	-1%	0	0	0.00	0.63	1%	0	0	0.00
	ember-cli	88.46	88.77	0%	0	0	0.00	89.02	1%	0	0	0.00
	hexo-server	0.90	0.91	1%	0	0	0.00	0.91	1%	0	0	0.00
	koop-core	1.48	1.49	0%	0	0	0.00	1.48	0%	0	0	0.00
	server	10.01	10.08	1%	0	0	0.00	10.07	1%	0	0	0.00

Results for clients of all projects with guarded execution mode enabled

	Client	Static Callgraph			Dynamic Callgraph		
Proj	**Client Proj**	**T(s)**	**SD**	**Exp.KB**	**T(s)**	**SD**	**Exp.KB**
memfs	artgen	10.46	7%	0	10.66	8%	0
	class-component-converter	13.65	29%	906.67	11.42	15%	906.67
	jaxon-ts	24.46	21%	906.67	22.38	14%	906.67
	webpack-dev-middleware	14.56	19%	906.67	14.41	18%	906.67
	zenobia-ts	22.33	17%	906.67	20.73	10%	906.67
fs-nextra	adset-DEPRECATED	0.34	0%	94.01	0.37	9%	0.00
	AdvancedJS	0.34	14%	0.00	0.31	7%	0.00
	better-klasa	0.37	27%	0.00	0.31	14%	0.00
	core	1.94	21%	94.01	1.53	0%	0.00
	klasa	2.1527	0%	94.01	2.65	19%	0.00
body-parser	appium-base-driver	10.48	17%	548.94	10.29	16%	21.92
	express	4.55	77%	740.02	3.05	66%	180.91
	karma	2.66	22%	614.78	2.35	12%	177.69
	moleculer-web	8.16	29%	740.02	6.52	11%	180.91
	typescript-rest	14.85	11%	740.02	14.83	11%	180.91
commander	html-minifier	10.20	11%	0.00	10.36	12%	0.00
	lint-staged	20.30	0%	0.00	20.56	2%	0.00
	metalsmith	1.43	1%	0.00	1.42	2%	0.00
	nunjucks	37.97	7%	0.00	37.84	7%	0.00
	sheetjs	23.31	1%	0.00	23.35	1%	0.00
memory-fs	astroturf	22.60	12%	252.45	21.95	9%	43.11
	mochapack	54.31	3%	247.20	59.42	12%	43.11
	rax	30.21	23%	0.00	25.77	9%	0.00
	vue-builder	2.84	30%	255.09	1.99	1%	43.11
	webpack	400.67	15%	0.00	403.21	16%	0.00
glob	copyfiles	1.53	23%	126.81	1.33	11%	32.36
	dot-object	1.84	-10%	111.15	1.80	12%	32.36
	node-glob-all	0.30	7%	100.19	0.28	3%	32.36
	replace-in-file	2.50	9%	122.88	2.31	2%	32.36
	stylus	3.07	13%	80.65	2.83	6%	32.36
redux	Choices	5.37	6%	59.42	5.38	6%	59.42
	found	37.78	19%	59.42	37.11	18%	59.42
	Griddle	9.36	5%	59.42	9.77	9%	59.42
	react-beautiful-dnd	63.86	3%	59.42	67.31	8%	59.42
	redux-ignore	0.57	1%	59.42	0.59	4%	59.42
css-loader	astroturf	20.23	9%	910.56	20.50	11%	3947.90
	custom-react-script	1.79	8%	0.00	1.74	5%	0.00
	docusaurus	142.66	5%	0.00	143.02	6%	0.00
	playroom	2.90	8%	0.00	2.88	8%	0.00
	powerbi-visual-tools	472.62	6%	0.00	496.00	11%	0.00
	decompress-zip	1.22	43%	240.87	0.78	10%	16.57
	downshift	1.47	3%	240.87	1.63	13%	3.16

q	node-ping	4.89	22%	240.87	4.69	19%	14.86
	passport-saml	0.48	13%	0.00	0.51	19%	0.00
	requestify	3.30	12%	240.87	3.43	15%	4.30
send	connect-gzip-static	0.71	27%	234.09	0.60	15%	12.68
	gitbook	5.00	2%	0.00	4.93	0%	0.00
	lasso	13.20	0%	0.00	13.49	2%	0.00
	node-restify	24.50	5%	240.59	24.60	6%	12.68
	serve-static	0.68	9%	258.33	0.68	8%	12.68
serve-favicon	appium-base-driver	8.97	7%	14.36	8.56	3%	0.00
	enb	4.16	7%	14.36	3.89	1%	0.00
	express-octoblu	0.91	11%	23.00	0.82	1%	0.00
	loopback	30.81	4%	14.36	30.11	2%	0.00
	server	10.55	6%	23.00	10.37	4%	0.00
morgan	appium-base-driver	9.32	8%	130.51	9.29	8%	10.49
	ember-cli	93.78	4%	141.86	92.52	3%	10.49
	gulp-contrib-connect	1.00	11%	130.51	1.01	12%	10.49
	json-server	23.34	15%	135.27	22.43	12%	10.49
	superstatic	1.01	7%	132.44	1.19	21%	10.49
serve-static	connect-gzip-static	0.64	13%	356.58	0.56	1%	12.87
	gulp-connect	0.74	7%	356.58	0.71	3%	12.87
	moleculer-web	6.50	2%	356.58	6.48	2%	12.87
	reload	9.82	19%	356.58	9.15	13%	12.87
	soap	1.41	-1%	0.00	1.55	8%	0.00
prop-types	react-dates	18.95	2%	116.81	22.79	18%	116.81
	react-loadable	53.70	3%	116.81	52.81	1%	116.81
	react-redux	7.49	6%	116.81	7.27	4%	116.81
	redux-form	16.29	5%	116.81	15.73	2%	116.81
	wd	20.43	12%	116.81	19.66	8%	116.81
compression	cordova-serve	0.72	14%	0.00	0.69	10%	0.00
	ember-cli	97.89	10%	0.00	96.75	9%	0.00
	hexo-server	1.02	12%	0.00	1.03	13%	0.00
	koop-core	1.70	13%	0.00	1.70	13%	0.00
	server	11.35	12%	0.00	11.28	11%	0.00

Example of dynamically loaded code when running in guarded execution mode

Consider this small demonstrative example: function `hello` takes a function as an argument and calls it with argument `args` (line 2740); `hello` is then called once with `console.log` (line 2743), and once with `eval` (line 2744).

```
2738 // file.js
2739 function hello( callback, arg) {
2740         callback(arg);
2741 }
2742
2743 hello( console.log, "hi");
2744 hello( eval, "danger!!")
```

If this code was part of an expanded stub, in guarded execution mode *every* function call is wrapped in a check to see if the function being called is one of the specified "dangerous" functions, that in this case are `eval`, `process.exec`, `child_process.spawn`, and `child_process.fork`. The code above gets transformed into:

```
2745 let dangerousFunctions = [eval];
2746 if( process){dangerousFunctions += [process.exec]};
2747 if( child_process){dangerousFunctions += [
     ↪ child_process.exec, child_process.fork,
     ↪ child_process.spawn]}
2748
2749 function hello(callback, arg) {
2750   (() => {
2751     let tempExp__uniqID = callback;
2752     if (dangerousFunctions.indexOf(tempExp__uniqID)
     ↪ > -1) console.warn("[STUBBIFIER] WARNING:
     ↪ Dangerous call in expansion of file.js");
2753     return callback(arg);
2754   })();
2755 }
2756
2757 (() => {
2758   let tempExp__uniqID = hello;
2759   if (dangerousFunctions.indexOf(tempExp__uniqID) >
     ↪ -1) console.warn("[STUBBIFIER] WARNING:
     ↪ Dangerous call in expansion of file.js");
2760   return hello(console.log, "hi");
2761 })();
2762
2763 (() => {
2764   let tempExp__uniqID = hello;
2765   if (dangerousFunctions.indexOf(tempExp__uniqID) >
     ↪ -1) console.warn("[STUBBIFIER] WARNING:
     ↪ Dangerous call in expansion of file.js");
2766   return hello(eval, "danger!!");
2767 })();
```

On line 2745 we see the array of "dangerous" functions is initialized, then expanded over the next few lines. Then, in the original code, every function call is wrapped in an Immediately Invoked Function Expression (IIFE), that:

- creates an alias to the function in question (`var tempExp__uniqID`)
- checks the array of dangerous functions to see if it contains this function, and if so prints a warning
- returns the call to the original function
- calls itself (since it is an IIFE)

Note that the call to the `eval` would be caught on line 2752, when it is passed as the parameter `callback` of function `hello`.

The purpose of the IIFE wrappers is to create closures, to avoid polluting the namespace of the program and enable us to use the same temporary variable name for every check.

Note: in our implementation, we design the guarded execution mode so that the original program behaviour is preserved modulo some warnings being printed. However, this would be easy to customize to (for example) throw an error, or exit the program, if a dangerous function call is encountered. The change would simply be to insert the desired code in place of the current `console.warn`.