Investigating and Recommending Co-Changed Entities for JavaScript Programs

Zijian Jiang[a], Hao Zhong[b], Na Meng[a,*]

[a] *Virginia Polytechnic Institute and State University, Blacksburg VA* 24060*, USA*
[b] *Shanghai Jiao Tong University, Shanghai* 200240*, China*

Abstract

JavaScript (JS) is one of the most popular programming languages due to its flexibility and versatility, but maintaining JS code is tedious and error-prone. In our research, we conducted an empirical study to characterize the relationship between co-changed software entities (e.g., functions and variables), and built a machine learning (ML)-based approach to recommend additional entity to edit given developers' code changes. Specifically, we first crawled 14,747 commits in 10 open-source projects; for each commit, we created at least one change dependency graph (CDG) to model the referencer-referencee relationship between co-changed entities. Next, we extracted the common subgraphs between CDGs to locate recurring co-change patterns between entities. Finally, based on those patterns, we extracted code features from co-changed entities and trained an ML model that recommends entities-to-change given a program commit.

According to our empirical investigation, (1) three recurring patterns commonly exist in all projects; (2) 80%–90% of co-changed function pairs either invoke the same function(s), access the same variable(s), or contain similar statement(s); (3) our ML-based approach CoRec recommended entity changes with high accuracy (73%–78%). CoRec complements prior work because it suggests changes based on program syntax, textual similarity, as well as software history; it achieved higher accuracy than two existing tools in our evaluation.

*Corresponding author
Email addresses: wz649588@vt.edu (Zijian Jiang), zhonghao@sjtu.edu.cn (Hao Zhong), nm8247@vt.edu (Na Meng)

Keywords: Multi-entity edit, change suggestion, machine learning, JavaScript

1. Introduction

JavaScript (JS) has become one of the most popular programming languages because it is lightweight, flexible, and powerful [1]. Developers use JS to build web pages and games. JS has many new traits (1) it is dynamic and weakly typed; (2) it has first-class functions; (3) it is a class-free, object-oriented programming language that uses prototypal inheritance instead of classical inheritance; and (4) objects in JS inherit properties from other objects directly and all these inherited properties can be changed at runtime. All above-mentioned traits make JS unique and powerful; they also make JS programs very challenging to maintain and reason about [2, 3, 4].

To reduce the cost of maintaining software, researchers proposed approaches that recommend code co-changes [5, 6, 7, 8]. For instance, Zimmermann et al. [5] and Rolfsnes et al. [6] mined co-change patterns of program entities from software version history and suggested co-changes accordingly. Wang et al. [7, 8] studied the co-change patterns of Java program entities and built CMSuggester to suggest changes accordingly for any given program commit. However, existing tools do not characterize any co-change patterns between JS software entities, neither do they recommend changes by considering the unique language features of JS or the mined co-changed patterns from JS programs (see Section 8.3) for detailed discussions).

To overcome the limitations of the prior approaches, in this paper, we first conducted a study on 14,747 program commits from 10 open-source JS projects to investigate (1) what software entities are usually edited together, and (2) how those simultaneously edited entities are related. Based on this characterization study for co-change patterns, we further developed a learning-based approach CoRec to recommend changes given a program commit.

Specifically in our study, for any program commit, we constructed and compared Abstract Syntax Trees (ASTs) for each edited JS file to identify all edited

entities (e.g., Deleted Classes (DC), Changed Functions (CF), and Added Variables (AV)). Next, we created change dependency graphs (CDGs) for each commit by treating edited entities as nodes and linking entities that have referencer-referencee relations. Afterwards, we extracted common subgraphs between CDGs and regarded those common subgraphs as recurring change patterns. In our study, we explored the following research question:

RQ1: What are the frequent co-change patterns in JS programs?

We automatically analyzed thousands of program commits from ten JS projects and revealed the recurring co-change patterns in each project. By manually inspecting 20 commits sampled for each of the 3 most popular patterns, we observed that 80%–90% of co-changed function pairs either invoke the same function(s), access the same variable(s), contain similar statement(s), or get frequently co-changed in version history.

Besides the above findings, our study reveals three most popular change patterns: (i) one or more caller functions are changed together with one changed callee function that they commonly invoke; (ii) one or more functions are changed together to commonly invoke an added function; (iii) one or more functions are changed together to commonly access an added variable. The co-changed callers in each pattern may share commonality in terms of variable accesses, function invocations, code similarity, or evolution history.

Based on the above-mentioned observations, we built a machine learning (ML)-based approach—CoRec—to recommend functions for co-change. Given the commits that contain matches for any of the above-mentioned co-change patterns, CoRec extracts 10 program features to characterize the co-changed function pairs, and uses those features to train an ML model. Afterwards, given a new program commit, the model predicts whether any unchanged function should be changed as well and recommends changes whenever possible. With CoRec, we investigated the following research question:

RQ2: How does CoRec perform when suggesting co-changes based on the observed three most popular patterns?

We applied CoRec and two existing techniques (i.e., ROSE [5] and Transitive Associate Rules (TAR) [9]) to the same evaluation datasets, and observed CoRec to outperform both techniques by correctly suggesting many more changes. CoRec's effectiveness varies significantly with the ML algorithm it adopts. CoRec works better when it trains three separate ML models corresponding to the three patterns than training a unified ML model for all patterns. Our results show that CoRec can recommend co-change functions with 73–78% accuracy; it significantly outperforms two baseline techniques that suggest co-changes purely based on software evolution.

We envision CoRec to be used in the integrated development environments (IDE) for JS, code review systems, and version control systems. In this way, after developers make code changes or before they commit edits to software repositories, CoRec can help detect and fix incorrectly applied multi-entity edits. In the sections below, we will first describe a motivating example (Section 2), and then introduce the concepts used in our research (Section 3). Next, we will present the empirical study to characterize co-changes in JS programs (Section 4). Afterwards, we will explain our change recommendation approach CoRec (Section 5) and expound on the evaluation results (Section 6). Our program and data are open sourced at: `https://github.com/NiSE-Virginia-Tech/wz649588-CoRec_jsAnalyzer`.

2. A Motivating Example

The prior work [10, 11, 12, 13] shows that developers may commit errors of omission (i.e., forgetting to apply edits completely) when they have to edit multiple program locations simultaneously in one maintenance task (i.e., bug fixing, code improvement, or feature addition). For instance, Fry et al. [10] reported that developers are over five times more precise at locating errors of commission than errors of omission. Yin et al. [12] and Park et al. [13] separately showed that developers introduced new bugs when applying patches to fix existing bugs. In particular, Park et al. inspected the supplementary bug

fixes following the initial bug-fixing trials, and summarized nine major reasons to explain why the initial fixes were incorrect. Two of the nine reasons were about the incomplete program edits applied by developers.

To help developers apply JS edits completely and avoid errors of omission, we designed and implemented a novel change recommendation approach—CoRec. This section overviews our approach with a running example, which is extracted from a program commit to Node.js—an open-source server-side JS runtime environment [14]. Figure 1 shows a simplified version of the exemplar program commit [15]. In this revision, developers added a function `maybeCallback(...)` to check whether the pass-in parameter `cb` is a function, and modified seven functions in distinct ways to invoke the added function(e.g., changing `fs.write(...)` on line 10 and line 14). The seven functions include: `fs.rmdir(...)`, `fs.appendFile(...)`, `fs.truncate(...)`, `fs.write(...)`, `fs.readFile(...)`, `fs.writeFile(...)`, and `fs.writeAll(...)` [15]. However, developers forgot to change an eighth function—`fs.read(...)`—to also invoke the added function (see line 19 in Figure 1).

```
1. + function maybeCallback(cb) {
2. +   return typeof cb === 'function' ? cb :
   rethrow();
3. + }

4.   fs.write = function(fd, buffer, offset,
   length, position, callback) {
5. -   callback =
   makeCallback(arguments[arguments.length - 1]);
6.     …
7.     req.oncomplete = wrapper;
8.     if (buffer instanceof Buffer) {
9.       …
10.+     callback = maybeCallback(callback);
11.      return binding.writeBuffer(…);
12.    }
13.    …
14.+   callback = maybeCallback(position);
15.    return binding.writeBuffer(fd, buffer,
   offset, …);
16.  }
```

```
17.  fs.read = function(fd, buffer, offset,
   length, position, callback) {
18.-   callback =
   makeCallback(arguments[arguments.length - 1]);
19.    …
   // an edit that developers forgot to apply:
   //+  callback = maybeCallback(callback);
20.    req.oncomplete = wrapper;
21.    binding.read(fd, buffer, offset, …);
22.  }
```

Figure 1: A program commit should add one function and change eight functions to invoke the newly added one. However, developers forgot to change one of the eight functions—`fs.read(...)` [15].

CoRec reveals the missing change with the following steps. CoRec first trains an ML model with the program co-changes extracted from Node.js software version history. Then given the exemplar commit, based on the added function

`maybeCallback(...)` and each changed function (e.g., `fs.write(...)`), CoRec extracts any commonality between the changed function and any unchanged one. For each function pair, CoRec applies its ML model to the extracted commonality features and predicts whether the function pair should be changed together. Because `fs.write(...)` and `fs.read(...)`

- commonly access one variable `binding`,
- commonly invoke two functions: `makeCallback(...)` and `wrapper(...)`,
- declare the same parameters in sequence,
- have token-level similarity as 41%, and
- have statement-level similarity as 42%,

the pre-trained ML model inside CoRec considers the two functions to share sufficient commonality and thus recommends developers to also change `fs.read(...)` to invoke `maybeCallback(...)`. In this way, CoRec can suggest entities for change, which edits developers may otherwise miss.

3. Terms and Definitions

This section first introduces concepts relevant to JS programming, and then describes the terminology used in our research.

ES6 and ES5. ECMA Script is the standardized name for JavaScript [16]. ES6 (or ECMAScript2015) is a major enhancement to ES5, and adds many features intended to make large-scale software development easier. ES5 is fully supported in all modern browsers, and major web browsers support some features of ES6. Our research is applicable to both ES5 and ES6 programs.

Software Entity. We use *software entity* to refer to any defined JS **class**, **function**, **variable**, or any **independent statement block** that is not contained by the definition of classes, functions, or variables. When developers write JS code, they can define each type of entities in multiple alternative ways. For instance, a class can be defined with a class expression (see Figure 2 (a)) or

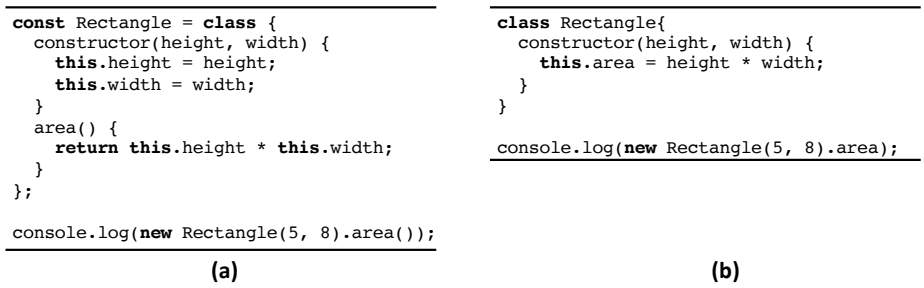


Figure 2: A JS class can be defined with an expression (see (a)) or a declaration (see (b)).

class declaration (see Figure 2 (b)). Similarly, a function can be defined with a function expression or function declaration. A variable can be defined with a variable declaration statement; the statement can either use keyword `const` to declare a constant variable, or use `let` or `var` to declare a non-constant variable.

Edited Entity. When maintaining JS software, developers may add, delete, or change one or more entities. Therefore, as with prior work [17], we defined a set of ***edited entities*** to describe the possible entity-level edits, including *Added Class* (**AC**), *Deleted Class* (**DC**), *Added Function* (**AF**), *Deleted Function* (**DF**), *Changed Function* (**CF**), *Added Variable* (**AV**), *Deleted Variable* (**DV**), *Changed Variable* (**CV**), *Added Statement Block* (**AB**), *Deleted Statement Block* (**DB**), and *Changed Statement Block* (**CB**). For example, if a new class is declared to have a constructor and some other methods, we consider the revision to have one AC, multiple AFs, and one or more AV (depending on how many fields are defined in the constructor).

Multi-Entity Edit and CDG. As with prior work [18], we use ***multi-entity edit*** to refer to any commit that has two or more ***edited entities***. We use ***change dependency graph (CDG)*** to visualize the the relationship between co-changed entities in a commit. Specifically, each CDG has at least two nodes and one edge. Each node represents an edited entity, and each edge represents the referencer-referencee relationship between entities (e.g., a function calls another function). Namely, if an edited entity E_1 refers to another edited entity E_2, we say E_1 depends on E_2. A related CDG is constructed to connect

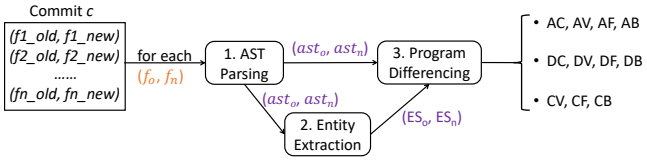


Figure 3: The procedure to extract changed entities given a commit.

the two entities with a directed edge pointing to E_2—the entity being depended upon (i.e. $E_1 \rightarrow E_2$). For each program commit, we may create zero, one, or multiple CDGs.

4. Characterization Study

This section introduces our study methodology (Section 4.1) and explains the empirical findings (Section 4.2). The purpose of this characterization study is to identify ***recurring change pattern (RCP)*** of JS programs. An RCP is a CDG subgraph that is commonly shared by the CDGs from at least two distinct commits. RCPs define different types of edits, and serve as the templates of co-change rules. Our approach in Section 5 mines concrete co-change rules for the most common RCPs.

4.1. Study Methodology

We implemented a tool to automate the analysis. Given a set of program commits in JS repositories, our tool first characterizes each commit by extracting the edited entities (Section 4.1.1) and constructing CDG(s) (Section 4.1.2). Next, it compares CDGs across commits to identify RCPs (Section 4.1.3).

4.1.1. Extraction of Edited Entities

As shown in Figure 3, we took three steps to extract any edited entities for each commit.

Step 1: AST Parsing. Given a program commit c, this step first locates the old and new versions of each edited JS file. For every edited file (f_o, f_n), this

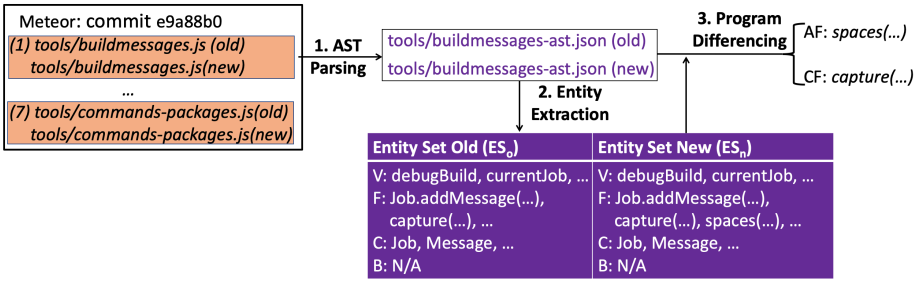


Figure 4: Extracting edited entities from a program commit of Meteor [21].

step adopts Esprima [19] and typed-ast-util [20] to generate Abstract Syntax Trees (ASTs): (ast_o, ast_n). Esprima is a high performance, standard-compliant JavaScript parser that supports the syntax of both ES5 and ES6; however, it cannot infer the static type binding information of any referenced class, function, or variable. Meanwhile, given JS files and the project's `package.json` file, typed-ast-util produces ASTs annotated with structured representations of TypeScript types, which information can facilitate us to precisely identify the referencer-referencee relationship between edited entities. We decided to use both tools for two reasons. First, when a project has `package.json` file, we rely on Esprima to identify the code range and token information for each parsed AST node, and rely on typed-ast-util to attach relevant type information to those nodes. Second, if a project has no `package.json` file, Esprima is still used to generate ASTs but we defined a heuristic approach (to be discussed later in Section 4.1.2) to identify the referencer-referencee relationship between entities with best efforts.

To facilitate our discussion, we introduce a working example from a program revision [21] of Meteor [22]. As shown in Figure 4, the program revision changes seven JS files. In this step, CoRec creates a pair of ASTs for each edited file and stores the ASTs into JSON files for later processing (e.g., `tools/buildmessages-ast.json (old)` and `tools/buildmessages-ast.json (new)`).

Step 2: Entity Extraction. From each pair of ASTs (ast_o, ast_n) (i.e., JSON files), this step extracts the entity sets (ES_o, ES_n). In the example shown in Figure 4, ES_o lists all entities from the old JS file, and ES_n corresponds to the

new file. We defined four kinds of entities to extract: variables (V), functions (F), classes (C), and statement blocks (B). A major technical challenge here is *how to extract entities precisely and consistently*. Because JS programming supports diverse ways of defining entities and the JS syntax is very flexible, we cannot simply check AST node types of statements to recognize entity definitions. For instance, a variable declaration statement can be interpreted as a variable-typed entity or a statement block, depending on the program context. To eliminate ambiguity and avoid any confusion between differently typed entities, we classify and extract entities in the following way:

- A code block is treated as a function definition if it satisfies either of the following two requirements. First, the AST node type is "`FunctionDeclaration`" (e.g., `runBenchmarks()` on line 7 in Figure 5) or "`MethodDefinition`". Second, (1) the block is either a "`VariableDeclaration`" statement (e.g., `const getRectArea = function(...){...};`) or an"`Assignment`" expression (see line 11 and line 20 of Figure 5); and (2) the right-side operand is either "`FunctionExpression`", or "`CallExpression`" that outputs another function as return value of the called function. In particular, if any defined function has its prototype property explicitly referenced (e.g., `Benchmark.prototype` on lines 20 and 24) or is used as a constructor to create any object (e.g., line 12), we reclassify the function definition as a class definition, because the function usage is more like the usage of a class.

- A code block is considered to be a class definition if it meets either of the following two criteria. First, the block uses keyword `class`. Second, the block defines a function, while the codebase either references the function's prototype (e.g., `Benchmark.prototype` on lines 20 and 24 in Figure 5) or uses the function as a constructor to create any object (see line 12).

- A code block is treated as a variable declaration if (1) it is either a "`VariableDeclaration`" statement (e.g., `var silent = ...` on line 2 in Figure 5) or an "`Assignment`" expression, (2) it does not define a function or

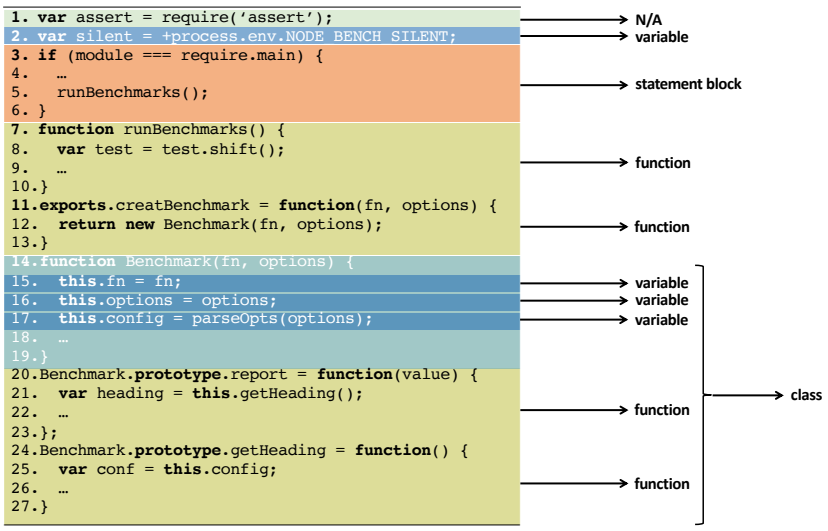


Figure 5: Code snippets from the file `benchmark.common.js` of Node.js in revision 00a1d36 [15], whose related entity types are shown on the right.

class, (3) it does not belong to the definition of any function but may belong to a constructor (see lines 15-17), and (4) it does not declare a required module (see line 1). Particularly, when a variable declaration is an assignment inside a class constructor (e.g., lines 15-17), it is similar to the field declaration in Java.

- A code block is treated as a statement block if (1) it purely contains statements, (2) it does not define any class, function, or variable, and (3) it does not belong to the definition of any class or function. For example, lines 3-6 in Figure 5 are classified as a statement block.

Step 3: Program Differencing. To identify any edited entity between ES_o and ES_n, we first matched the definitions of functions, variables, and classes across entity sets based on their signatures. If any of these entities (e.g., a function definition) solely exists in ES_o, an entity-level deletion (e.g., DF) is inferred; if an entity (e.g., a variable definition) solely exists in ES_n, an entity-level insertion (e.g., AV) is inferred. Next, for each pair of matched entities,

we further exploited a fine-grained AST differencing tool—GumTree [23]—to identify expression-level and statement-level edits. If any edit is reported, we inferred an entity-level change (e.g., CF shown in Figure 4). Additionally, we matched statement blocks across entity sets based on their string similarities. Namely, if a statement block $b_1 \in ES_o$ has the longest common subsequence with a block $b_2 \in ES_n$ and the string similarity is above 50%, we considered the two blocks to match. Furthermore, if the similarity between two matched blocks is not 100%, we inferred a block-level change CB.

4.1.2. CDG Construction

For each program commit, we built CDGs by representing the edited entities as nodes, and by connecting edited entities with directed edges if they have either of the following two kinds of relationship:

- **Access.** If an entity E_1 accesses another entity E_2 (i.e., by reading/writing a variable, invoking a function, or using a class), we consider E_1 to be dependent on E_2.
- **Containment.** If the code region of E_1 is fully covered by that of E_2, we consider E_1 to be dependent on E_2.

The technical challenge here is how to identify the relationship between edited entities. We relied on ESprima's outputs to compare code regions between edited entities in order to reveal the containment relations. Additionally, when `package.json` file is available, we leveraged the type binding information inferred by typed-ast-util to identify the access relationship. For instance, if there is a function call `bar()` inside an entity E_1 while `bar()` is defined by a JS module `f2`, then typed-ast-util can resolve the fully qualified name of the callee function as `f2.bar()`. Such resolution enables us to effectively link edited entities no matter whether they are defined in the same module (i.e., JS file) or not.

Since some JS projects have no `package.json` file, we could not adopt typed-ast-util to resolve bindings in such scenarios. Therefore, we also built a simpler

but more applicable approach to automatically speculate the type binding information of accessed entities as much as possible. Specifically, suppose that file `f1` defines E_1 to access E_2. To resolve E_2 and link E_1 with E_2's definition, this intuitive approach first scans all entities defined in `f1` to see whether there is any E_2 definition locally. If not, this approach further examines all `require` and `import` statements in `f1`, and checks whether any required or imported module defines a corresponding entity with E_2's name; if so, this approach links E_1 with the retrieved E_2's definition.

Compared with typed-ast-util, our approach is less precise because it cannot infer the return type of any invoked function. For instance, if we have `const foo = bar()` where `bar()` returns a function, our approach simply assumes `foo` to be a variable instead of a function. Consequently, our approach is unable to link `foo`'s definition with any of its invocations. Based on our experience of applying both typed-ast-util and the heuristic method to the same codebase (i.e., nine open-source projects), the differences between these two methods' results account for no more than 5% of all edited entities. It means that our heuristic method is still very precise even though no `package.json` file is available.

Figures 6 and 7 separately present the code changes and CDG related to `tools/buildmessage.js`, an edited file mentioned in Figure 4. According to Figure 6, the program commit modifies file `tools/buildmessage.js` by defining a new function `spaces(...)` and updating an existing function `capture(...)` to invoke the new function. It also changes file `tools/commands-package.js` by updating the function invocation of `capture(...)` inside a statement block (i.e., `main.registerCommand(...);`). Given the old and new versions of both edited JS files, our approach can construct the CDG shown in Figure 7. In this CDG, each directed edge starts from a dependent entity E_1, and points to the entity on which E_1 depends. Each involved function, variable, or class has its fully qualified name included in the CDG for clarity. As statement blocks have no fully qualified names, we created a unique identifier for each block with (1) the module name (e.g., `tools.commands-packages`) and (2) index of the block's first character in that module (e.g., `69933`).

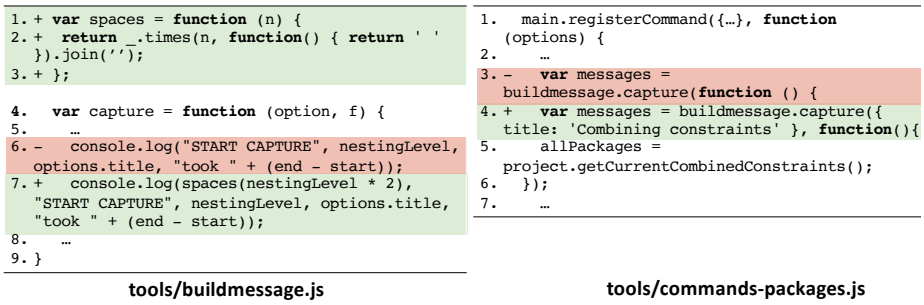


Figure 6: A simplified program commit that adds a function `spaces(...)`, changes a function `capture(...)`, and changes a statement block [21]

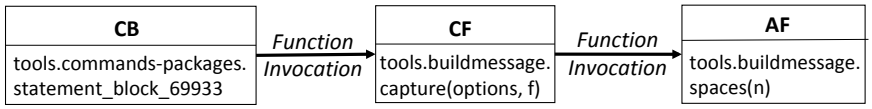


Figure 7: The CDG corresponding to the program commit shown in Figure 6

4.1.3. Extraction of Recurring Change Patterns (RCP)

As with prior work [18], we extracted RCPs by comparing CDGs across program commits. Intuitively, given a CDG g_1 from commit c_1 and the CDG g_2 from commit c_2, we matched nodes based on their edit-entity labels (e.g., AF) while ignoring the concrete code details (e.g., `tools.buildmessage.spaces(n)` in Figure 7). We then established edge matches based on those node matches. Namely, two edges are matched only if they have matching starting nodes and matching ending nodes. Next, based on all established matches, we identified the largest common subgraph between g_1 and g_2 using the off-the-shelf subgraph isomorphism algorithm VF2 [24]. Such largest common subgraphs are considered as RCPs because they commonly exist in CDGs of different commits.

4.2. Empirical Findings

To characterize JS code changes, we applied our study approach to a subset of available commits in 10 open-source projects, as shown in Table 1. We chose these projects because (1) they are popularly used; (2) they are from different application domains; and (3) they contain a large number of available

Table 1: Subject projects

Project	**Description**	**# of KLOC**	**# of Commits**	**# of Edited Entities**
Node.js	Node.js [14] is a cross-platform JS runtime environment. It executes JS code outside of a browser.	1,755	2,701	11,287
Meteor	Meteor [22] is an ultra-simple environment for building modern web applications.	255	3,011	10,274
Ghost	Ghost [25] is the most popular open-source and headless Node.js content management system (CMS) for professional publishing.	115	1,263	5,142
Habitica	Habitica [26] is a habit building program that treats people's life like a Role Playing Game.	129	1,858	6,116
PDF.js	PDF.js [27] is a PDF viewer that is built with HTML5.	104	1,754	4,255
React	React [28] is a JS library for building user interfaces.	286	1,050	4,415
Serverless	Serverless [29] is a framework used to build applications comprised of microservices that run in response to events.	63	1,110	3,846
Webpack	Webpack [30] is a module bundler, which mainly bundles JS files for usage in a browser. assets.	37	1,099	3,699
Storybook	Storybook [31] is a development environment for UI components.	43	528	2,277
Electron	Electron [32] is a framework that supports developers to write cross-platform desktop applications using JS, HTML, and CSS.	35	673	1,898

commits. In particular, we found these 10 projects by (1) ranking the JS repositories available on GitHub based on star counts, (2) picking the projects with lots of commits (>10,000 commits), and (3) focusing on the projects compliant with ES5 or ES6—the two mainstream syntax definitions followed by developers when they write JS code. For simplicity, to sample the commits that may fulfill independent maintenance tasks, we searched each software repository for commits whose messages contain any of the following keywords: "bug", "fix", "error", "adjust", and "failure".

Table 1 shows the statistics related to the sampled commits. In particular, column **# of KLOC** presents the code size of each project (i.e., the number of kilo lines of code (KLOC)). Column **# of Commits** reports the number of commits identified via our keyword-based search. Column **# of Edited Entities** reports the number of edited entities extracted from those sampled commits. According to this table, the code size of projects varies significantly from 35 KLOC to 1755 KLOC. Among the 10 projects, 528–3,011 commits were sampled, and 1,898–11,287 edited entities were included for each project. Within these projects, only Node.js has no `package.json` file, so we adopted our heuristic approach mentioned in Section 4.1.2 to link edited entities. For the remaining nine projects, as they all have `package.json` files, we leveraged the type binding information inferred by typed-util-ast to connect edited entities.

4.2.1. Commit Distributions Based on The Number of Edited Entities

We first clustered commits based on the number of edited entities they contain. Because the commit distributions of different projects are very similar to each other, we present the distributions for four projects in Figure 8. Among the 10 projects, 41%–52% of commits are multi-entity edits. Specifically, 15%–19% of commits involve two-entity edits, and 7%–10% of commits are three-entity edits. The number of commits decreases as the number of edited entities increases. The maximum number of edited entities appears in Node.js, where a single commit modifies 335 entities. We manually checked the commit on GitHub [33], and found that four JS files were added and three other JS files

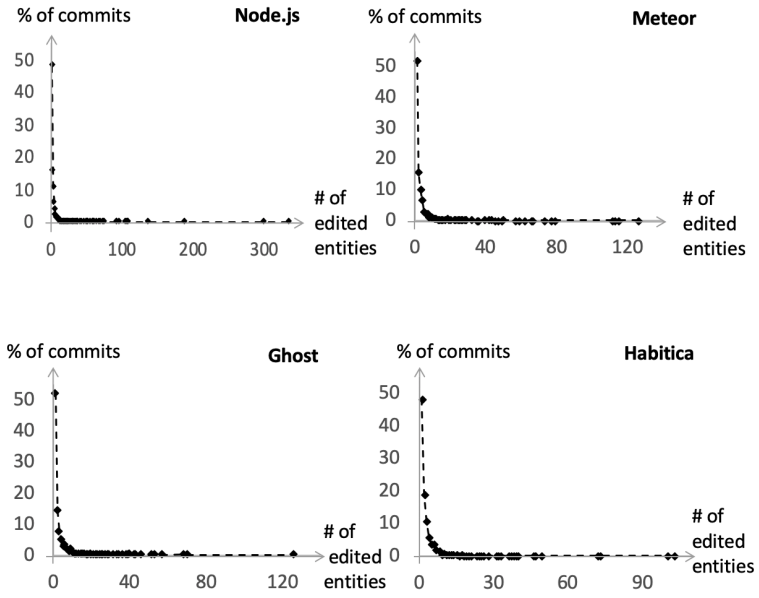


Figure 8: Commit distributions based on the number of edited entities each of them contains

were changed to implement HTTP/2.

We noticed that about half of sampled program revisions involve multi-entity edits. This observation implies the importance of co-change recommendation tools. When developers have to frequently edit multiple entities simultaneously to achieve a single maintenance goal, it is crucially important to provide automatic tool support that can check for the completeness of code changes and suggest any missing change when possible. In order to build such tools, we decided to further explore relations between co-changed entities (see Section 4.2.2).

> **Finding 1:** *Among the 10 studied projects, 41–52% of studied commits are multi-entity edits. It indicates the necessity of our research to characterize multi-entity edits and to recommend changes accordingly.*

4.2.2. Commit Distributions Based on The Number of CDGs

We further clustered multi-entity edits based on the number of CDGs constructed for each commit. As shown in Table 2, our approach created CDGs for

Table 2: Multi-entity edits and created CDGs

Project	# of Multi-Entity Edits	# of Multi-Entity Edits with CDG(s) Extracted	% of Multi-Entity Edits with CDG(s) Extracted
Node.js	1,401	785	56%
Meteor	1,445	670	46%
Ghost	604	356	59%
Habitica	962	345	36%
PDF.js	711	372	52%
React	538	320	60%
Serverless	480	171	36%
Webpack	483	253	52%
Storybook	243	119	49%
Electron	277	123	44%

36–60% of the multi-entity edits in distinct projects. On average, 49% of multi-entity edits contain at least one CDG. Due to the complexity and flexibility of the JS programming language, it is very challenging to statically infer all possible referencer-referencee relationship between JS entities. Therefore, the actual percentage of edits that contain related co-changed entities can be even higher than our measurement. Figure 9 presents the distributions of multi-entity edits based on the number of CDGs extracted. Although this figure only presents the commit distributions for four projects: Node.js, Meteor, Ghost, and Habitica, we observed similar distributions in other projects as well. As shown in this figure, the number of commits decreases significantly as the number of CDGs increases. Among all 10 projects, 73%–81% of commits contain single CDGs, 9%–18% of commits have two CDGs extracted, and 3%–7% of commits have three CDGs each. The commit with the largest number of CDGs constructed (i.e., 16) is the one with the maximum number of edited entities in Node.js as mentioned above in Section 4.2.1.

The high percentage of multi-entity edits with CDGs extracted (i.e., 49%) implies that JS programmers usually change syntactically related entities simul-

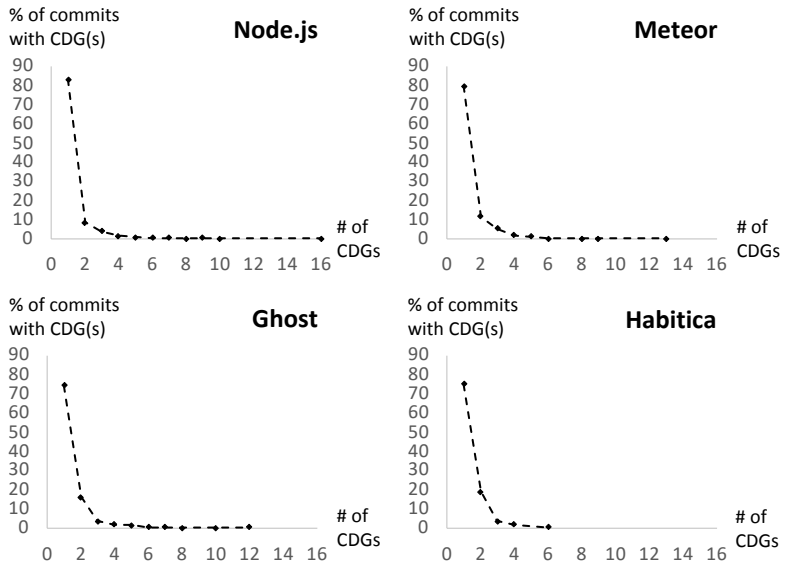


Figure 9: The distributions of multi-entity edits based on the number of CDGs

taneously in program revisions. Such syntactic relevance between co-changed entities enlightened us to build tools that recommend co-changes by observing the syntactic dependences between changed and unchanged program entities. To concretize our approach design for co-change recommendations, we further explored the recurring syntactic relevance patterns between co-changed entities, i.e., RCPs (see Section 4.2.3).

Finding 2: *For 36–60% of multi-entity edits in the studied projects, our approach created at least one CDG for each commit. It means that many simultaneously edited entities are syntactically relevant to each other.*

4.2.3. Identified RCPs

By comparing CDGs of distinct commits within the same project repository, we identified RCPs in all projects. As listed in Table 3, 32–205 RCPs are extracted from individual projects. In each project, there are 113–782 commits that contain matches for RCPs. In particular, each project has 228–2,385 subgraphs matching RCPs. By comparing this table with Table 2, we found that

Table 3: Recurring change patterns (RCPs) and their matches

Projects	**# of RCPs**	**# of Commits with RCP Matches**	**# of Subgraphs Matching the RCPs**
Node.js	205	782	2,385
Meteor	182	658	1,719
Ghost	123	351	1,223
Habitica	102	339	706
PDF.js	76	367	640
React	101	316	899
Serverless	52	164	372
Webpack	73	243	583
Storybook	38	113	337
Electron	32	117	228

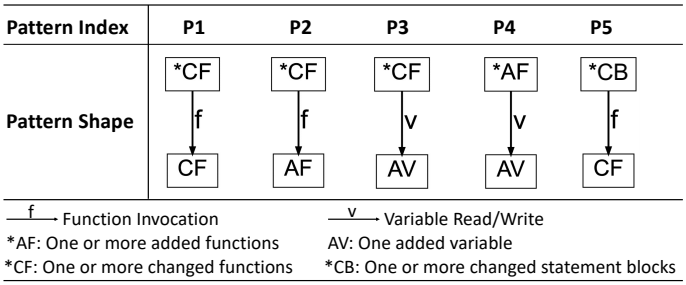


Figure 10: The 5 most popular recurring change patterns among the 10 projects

95%–100% of the commits with CDGs extracted have matches for RCPs. It means that if one or more CDGs can be built for a commit, the commit is very likely to share common subgraphs with some other commits. In other words, simultaneously edited entities are usually correlated with each other in a fixed number of ways. If we can characterize the frequently occurring relationship between co-changed entities, we may be able to leverage such characterization to predict co-changes or reveal missing changes.

By accumulating the subgraph matches for RCPs across projects, we identified five most popular RCPs, as shown in Figure 10. Here, **P1** means that

Table 4: The numbers of RCPs revealed by different threshold settings

Projects	**≥2 commits**	**≥3 commits**	**≥4 commits**	**≥5 commits**
Node.js	205	174	155	118
Meteor	182	145	120	104
Ghost	123	110	91	78
Habitica	102	69	53	43
PDF.js	76	61	56	40
React	101	78	65	53
Serverless	52	35	31	24
Webpack	73	58	43	38
Storybook	38	21	19	17
Electron	32	28	21	16

when a callee function is changed, one or more of its caller functions are also changed. **P2** means that when a new function is added, one or more existing functions are changed to invoke that new function. **P3** shows that when a new variable is added, one or more existing functions are changed to read/write the new variable. **P4** presents that when a new variable is added, one or more new functions are added to read/write the new variable. **P5** implies that when a function is changed, one or more existing statement blocks invoking the function are also changed. Interestingly, the top three patterns commonly exist in all 10 projects, while the other two patterns do not exist in some of the projects. The top three patterns all involve simultaneously changed functions.

The threshold used to identify RCPs. By default, we detected an RCP if the CDGs of at least **two** commits share any subgraph. Alternatively, we can also set this threshold for the number of commits sharing RCP(s) to any number beyond 2 (e.g., 3, 4, and 5). To study how the threshold setting affects our RCP observations, we conducted an experiment to set the value to 3, 4, and 5 separately. Table 4 presents the experiment results. Unsurprisingly, as the threshold increases, the number of revealed RCPs decreases because fewer RCPs can satisfy the requirement on commit counts. Specifically for Meteor,

the threshold setting 2 leads to 182 RCPs found, while the threshold setting 5 causes 104 RCPs detected. Although the total number of RCPs decreases as the threshold increases, we found the most popular five RCPs to remain the same. To reveal as many RCPs as possible, by default, we set the threshold to 2.

> **Finding 3:** *Among the commits with CDGs extracted, 95%–100% of commits have matches for mined RCPs. In particular, the most popular three RCPs all involve simultaneously changed functions.*

4.2.4. Case Studies for The Three Most Popular RCPs

We conducted two sets of case studies to understand (1) the semantic meanings of P1–P3 and (2) any co-change indicators within code for those patterns. In each set of case studies, we randomly sampled 20 commits matching each of these RCPs and manually analyzed the code changes in all 60 commits.

The Semantic Meanings of P1–P3. In the 20 commits sampled for each pattern, we summarized the semantic relevance of entity-level changes as below.

Observations for P1 (*CF$\xrightarrow{f}$CF). We found the caller and callee functions changed together in three typical scenarios. First, in about 45% of the inspected commits, both caller and callee functions experienced *consistent changes* to invoke the same function(s), access the same variable(s), or execute the same statement(s). Second, in about 40% of the commits, developers applied *adaptive changes* to callers when callees were modified. The adaptive changes involve modifying caller implementations when the signatures of callee functions were updated, or moving code from callees to callers. Third, in 15% of cases, callers and callees experienced seemingly irrelevant changes.

Observations for P2 (*CF$\xrightarrow{f}$AF). Such changes were applied for two major reasons. First, in 65% of the studied commits, the added function implemented some new logic, which was needed by the changed caller function. Second, in the other 35% of cases, changes were applied for refactoring purposes. Namely, the added function was extracted from one or more existing functions and those functions were simplified to just invoke the added function.

Observations for P3 (*CF$\xrightarrow{v}$AV). Developers applied such changes in two typical scenarios. First, in 60% of cases, developers added a new variable for feature addition, which variable was needed by each changed function (i.e., cross-cutting concern [34]). Second, in 40% of the cases, developers added variables for refactoring purposes. For instance, some developers added a variable to replace a whole function, so all caller functions of the replaced function were consistently updated to instead access the new variable. Additionally, some other developers added a variable to replace some expression(s), constant(s), or variable(s). Consequently, the functions related to the replaced entities were consistently updated for the new variable.

The Code Indicators for Co-Changes in P1–P3. To identify potential ways of recommending changes based on the mined RCPs, we randomly picked 20 commits matching each pattern among P1–P3; we ensured that each sampled commit has two or more co-changed functions (e.g., *CF) referencing another edited entity. We then inspected the co-changed functions in each commit, to decide whether they share any commonality that may indicate their simultaneous changes. As shown in Table 5, the three case studies I–III correspond to the three patterns P1–P3 in sequence. In our manual analysis, we mainly focused on four types of commonality:

- **FI**: The co-changed functions commonly invoke one or more *peer functions* of the depended entity E (i.e., CF in P1, AF in P2, and AV in P3). Here, **peer function** is any function that is defined in the same file as E.
- **VA**: The co-changed functions commonly access one or more *peer variables* of the depended entity E. Here, **peer variable** is any variable that is defined in the same file as E.
- **ST**: The co-changed functions commonly share at least 50% of their token sequences. We calculated the percentage with the longest common subsequence algorithm between two token strings.

- **SS**: The co-changed functions commonly share at least 50% of their statements. We computed the percentage by recognizing identical statements between two given functions `f1(...)` and `f2(...)`. Assume that the two functions separately contain n_1 and n_2 statements, and the number of common statements is n_3. Then the percentage is calculated as

$$\frac{n_3 \times 2}{n_1 + n_2} \times 100\% \quad (1)$$

Table 5: Commonality observed between the co-changed functions

Case Study	Commonality				No Commonality
	FI	VA	ST	SS	
I (for P1: *CF $\xrightarrow{f}$ CF)	8	5	7	4	4
II (for P2: *CF $\xrightarrow{f}$ AF)	12	7	8	6	2
III (for P3: *CF $\xrightarrow{v}$ AV)	6	13	6	5	3

We will define more commonalities between co-changed functions in Section 5.2 and Table 6. We studied the above-mentioned four kinds of commonality for two reasons. First, **ST** and **SS** capture the typical textual/syntactic similarity between functions; Second, **FI** and **VA** reflect the typical semantic similarity between functions (i.e., invoking the same functions or accessing the same variables). We were inspired to check these four types of commonality by prior research on recommendation systems [35, 36, 37] and code clones [38, 39, 40]. Each of these inspiring papers is relevant to identifying similarity between program entities. For instance, CCFinder [38] explores and adopts three types of commonality: similar token sequences, common variable usage, and common function calls.

According to Table 5, 80%–90% of co-changed functions share certain commonality with each other. There are only 2–4 commits in each study where the co-changed functions share nothing in common. Particularly, in the first case study, the FI commonality exists in eight commits, VA exists in five commits, ST occurs in seven commits, and SS occurs in four commits. The summation of these commonality occurrences is larger than 20, because the co-changed functions in some commits share more than one type of commonality. Additionally,

the occurrence rates of the four types of commonality are different between case studies. For instance, FI has 8 occurrences in the first case study; it occurs in 12 commits of the second study and occurs in only 6 commits of the third study. As another example, most commits (i.e., 13) in the third study share the VA commonality, while there are only 5 commits in the first study having such commonality. The observed differences between our case studies imply that when developers apply multi-entity edits matching different RCPs, the commonality shared between co-changed functions also varies.

Finding 4: *When inspecting the relationship between co-changed functions in three case studies, we found that these functions usually share certain commonality. This indicates great opportunities for developing co-change recommendation approaches.*

5. Our Change Recommendation Approach: CoRec

In our characterization study (see Section 4), we identified three most popular RCPs: *CF$\xrightarrow{f}$CF, *CF$\xrightarrow{f}$AF, and *CF$\xrightarrow{v}$AV. In all these patterns, there is at least one or more changed functions (i.e., *CF) that references another edited entity E (i.e., CF, AF, or AV). In the scenarios when two or more co-changed functions commonly depend on E, we also observed certain commonality between those functions. This section introduces our recommendation system—CoRec—which is developed based on the above-mentioned insights. As shown in Figure 11, CoRec has three phases. In the following subsections (Sections 5.1-5.3), we explain each phase in detail.

5.1. Phase I: Commit Crawling

Given the repository of a project P, Phase I crawls commits to locate any data usable for machine learning. Specifically, for each commit in the repository, this phase reuses part of our study approach (see Sections 4.1.1 and 4.1.2) to extract edited entities and to create CDGs. If a commit c has any subgraph

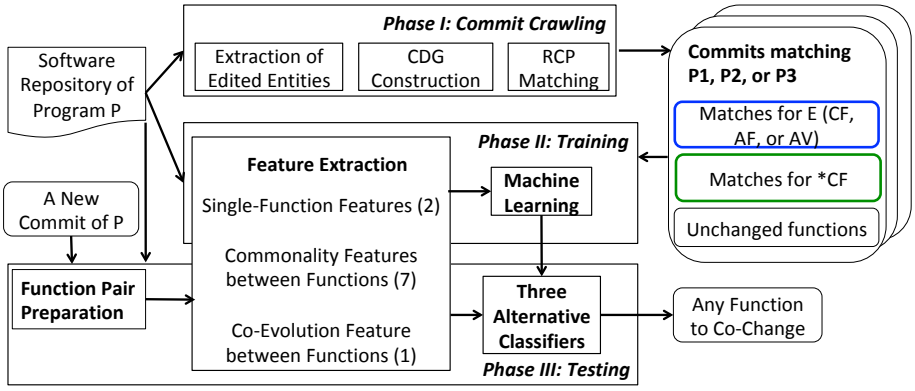


Figure 11: CoRec consists of three phases: commit crawling, training, and testing

matching P1, P2, or P3, this phase recognizes the entity E_m matching E (i.e., an entity matching CF in P1, matching AF in P2, or matching AV in P3) and any co-changed function matching *CF. We denote these co-changed function(s) with $CF\_Set = \{cf_1, cf_2, \ldots\}$, and denote the unchanged function(s) in edited JS files from the same commit with $UF\_Set = \{uf_1, uf_2, \ldots\}$. If CF_Set has at least two co-changed functions, CoRec considers the commit to be usable for model training and passes E_m, CF_Set, and UF_Set to the next phase.

5.2. Phase II: Training

This phase has two major inputs: the software repository of program P, and the extracted data from each relevant commit (i.e., E_m, CF_Set, and UF_Set). In this phase, CoRec first creates positive and negative training samples, and then extracts features for each sample. Next, CoRec trains a machine learning model by applying Adaboost (with Random Forest as the "weak learner") [41] to the extracted features. Specifically, to create positive samples, CoRec enumerates all possible function pairs in CF_Set, because each pair of these functions were co-changed with E_m. We represent the positive samples with $Pos = \{(cf_1, cf_2), (cf_2, cf_1), (cf_1, cf_3), \ldots\}$. To create negative samples, CoRec pairs up each changed function $cf \in CF\_Set$ with an unchanged function $uf \in UF\_Set$, because each of such function pairs were not co-changed. Thus,

Table 6: A list of features extracted for function pair (f_1, f_2)

Id	Feature	Id	Feature
1	Number of E_m-relevant parameter types in f_2	6	Whether f_1 and f_2 have the same return type
2	Whether f_2 has the E_m-related type	7	Whether f_1 and f_2 are defined in the same way
3	Number of common peer variables	8	Token similarity
4	Number of common peer functions	9	Statement similarity
5	Number of common parameter types	10	Co-evolution frequency

we denote the negative samples as $Neg = \{(cf_1, uf_1), (uf_1, cf_1), (cf_1, uf_2), \ldots\}$. By preparing positive and negative samples in this way, given certain pair of functions, we expect the trained model to predict whether the functions should be co-changed or not.

CoRec extracts 10 features for each sample. As illustrated in Figure 11, two features reflect code characteristics of the second function in the pair, seven features capture the code commonality between functions, and one feature focuses on the co-evolution relationship between functions. Table 6 presents more details of each feature. Specifically, the 1^{st} and 2^{nd} features are about the relationship between f_2 and E_m. Their values are calculated as below:

- When E_m is CF or AF, the 1^{st} feature records the number of types used in f_2 that match any declared parameter type of E_m. Intuitively, the more type matches, the more likely that f_2 should be co-changed with E_m. The 2^{nd} feature checks whether the code in f_2 uses the return type of E_m.

- When E_m is AV, the 1^{st} feature is set to zero, because there is no parameter type involved in variable declaration. The 2^{nd} feature checks whether the code in f_2 uses the data type of the newly added variable.

The 3^{rd} and 4^{th} features were calculated in similar ways. Specifically, depending on which JS file defines E_m, CoRec locates peer variables (i.e., variables defined within the same file as E_m) and peer functions (i.e., functions defined in the same file). Next, CoRec identifies the accessed peer variables (or peer func-

tions) by each function in the pair, and intersects the sets of both functions to count the commonly accessed peer variables (or peer functions). Additionally, the 7^{th} feature checks whether f_1 and f_2 are defined in the same manner. In our research, we consider the following five ways to define functions:

(1) via `FunctionDeclaration`, e.g., `function foo(...){...}`,

(2) via `VariableDeclaration`, e.g., `var foo = function(...){...}`,

(3) via `MethodDefinition`, e.g., `Class A {foo(...){...}}`,

(4) via `PrototypeFunction` to extend the prototype of an object or a function, e.g., `x.prototype.foo = function(...){...}`, and

(5) via certain `exports`-related statements, e.g., `exports.foo = function(...){...}` and `module.exports = {foo: function(...){...}}`.

If f_1 and f_2 are defined in the same way, the 7^{th} feature is set to `true`. Finally, the 10^{th} feature assesses in the commit history, how many times the pair of functions were changed together before the current commit. Inspired by prior work [5], we believe that the more often two functions were co-changed in history, the more likely that they are co-changed in the current or future commits.

Depending on the type of E_m, CoRec takes in extracted features to actually train three independent classifiers, with each classifier corresponding to one pattern among P1–P3. For instance, one classifier corresponds to P1: *CF$\xrightarrow{f}$CF. Namely, when E_m is CF and one of its caller functions cf is also changed, this classifier predicts whether there is any unchanged function uf that invokes E_m and should be also changed. The other two classifiers separately predict functions for co-change based on P2 and P3. We consider these three binary-class classifiers as an integrated machine learning model, because all of them can take in features from one program commit and related software version history, in order to recommend co-changed functions when possible.

5.3. Phase III: Testing

This phase takes in two inputs—a new program commit c_n and the related software version history, and recommends any unchanged function that should have been changed by that commit. Specifically, given c_n, CoRec reuses the steps of Phase I (see Section 5.1) to locate E_m, CF_Set, and UF_Set. CoRec then pairs up every changed function $cf \in CF\_Set$ with every unchanged one $uf \in UF\_Set$, obtaining a set of candidate function pairs $Candi = \{(cf_1, uf_1), (uf_1, cf_1), (cf_1, uf_2), \ldots\}$. Next, CoRec extracts features for each candidate p and sends the features to a pre-trained classifier depending on E_m's type. If the classifier predicts the function pair to have co-change relationship, CoRec recommends developers to also modify the unchanged function in p.

6. Evaluation

In this section, we first introduce our experiment setting (Section 6.1) and the metrics used to evaluate CoRec's effectiveness (Section 6.2). Then we explain our investigation with different ML algorithms and present CoRec's sensitivity to the adopted ML algorithms (Section 6.3), through which we finalize the default ML algorithm applied in CoRec. Next we expound on the effectiveness comparison between CoRec and two existing tools: ROSE [5] and Transitive Associate Rules (TAR) [9] (Section 6.4). Finally, we present the comparison between CoRec and a variant approach that trains one unified classifier instead of three distinct classifiers to recommend changes (Section 6.5).

6.1. Experiment Setting

We mined repositories of the 10 open-source projects introduced in Section 4, and found three distinct sets of commits in each project that are potentially usable for model training and testing. As shown in Table 7, in total, we found 280 commits matching P1, 232 commits matching P2, and 182 commits matching P3. Each of these pattern matches has at least two co-changed functions (*CF) depending on E_m. In our evaluation, for each data set of each project, we could

Table 7: Numbers of commits that are potentially usable for model training and testing

Project	**# of Commits Matching P1**	**# of Commits Matching P2**	**# of Commits Matching P3**
Node.js	92	77	65
Meteor	67	59	39
Ghost	21	24	28
Habitica	11	8	5
PDF.js	14	12	14
React	18	12	5
Serverless	26	12	8
Webpack	22	24	8
Storybook	2	1	4
Electron	7	3	6
Sum	**280**	**232**	**182**

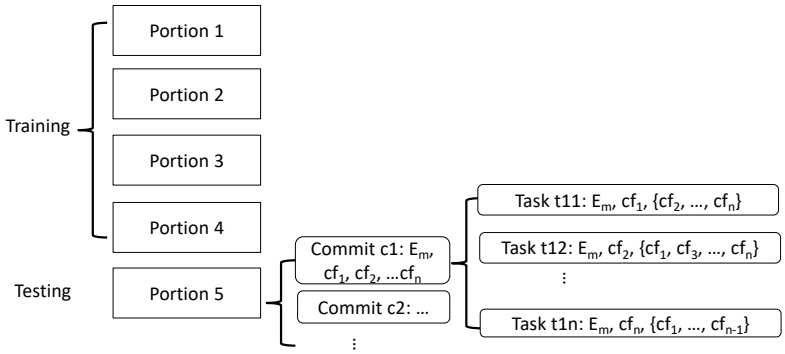


Figure 12: Typical data processing for each fold of the five-fold cross validation

use part of the data to train a classifier and use the remaining data to test the trained classifier. Because Storybook and Electron have few commits, we excluded them from our evaluation and simply used the identified commits of the other eight projects to train and test classifiers.

We decided to use k-fold cross validation to evaluate CoRec's effectiveness. Namely, for every data set of each project, we split the mined commits into k portions roughly evenly; each fold uses $(k-1)$ data portions for training and the remaining portion for testing. Among the eight projects, because each project has at least five commits matching each pattern, we set $k = 5$ to diversify our

evaluation scenarios as much as possible. For instance, Habitica has five commits matching P3. When evaluating CoRec's capability of predicting co-changes for Hibitica based on P3, in each of the five folds, we used four commits for training and one commit for testing. Figure 12 illustrates our five-fold cross validation procedure. In the procedure, we ensured that each of the five folds adopted a distinct data portion to construct prediction tasks for testing purposes. For any testing commit that has n co-changed functions (*CF) depending on E_m, i.e., $CF\_Set = \{cf_1, cf_2, \ldots, cf_n\}$, we created n prediction tasks in the following way. In each prediction task, we included one known changed function cf_i ($i \in [1, n]$) together with E_m and kept all the other $(n-1)$ functions unchanged. We regarded the $(n-1)$ functions as ground truth (GT) to assess how accurately CoRec can recommend co-changes given E_m and cf_i.

For instance, one prediction task we created in React includes the followings: $E_m =$ `src/isomorphic/classic/types.ReactPropTypes.createChainableTypeChecker(...)`, $cf =$ `src/isomorphic/classic/types.ReactPropTypes.createObjectOfTypeChecker(...)`, and $GT =\{$`src/isomorphic/ classic/types.ReactPropTypes.createShapeTypeChecker(...)`$\}$. When CoRec blindly pairing cf with any unchanged function, it may extract feature values as below: feature1 = 1, feature2 = `true`, feature3 = 0, feature4 = 2, feature5 = 0, feature6 = `true`, feature7 = `true`, feature8 = 76%, feature9 = 45%, feature10 = 1}. Table 8 shows the total numbers of prediction tasks we created for all projects and all patterns among the five-fold cross validation.

Notice that our five-fold cross validation is different from the leave-one-out (LOO) cross validation. With LOO, given N prediction tasks, we need to use $(N-1)$ tasks for training and 1 task for testing, and to repeat the experiment N times. However, in our five-fold cross validation, each commit corresponds to a different number of tasks (see Table 8). Suppose that we are given five commits in total, and each commit corresponds to n_1, n_2, ..., or n_5 tasks. When training a model using the first four commits and testing that model with the fifth one, we actually use $(n_1 + n_2 + n_3 + n_4)$ tasks for training and n_5 tasks for testing.

Table 8: Total numbers of prediction tasks involved in the five-fold cross validation

Project	**# of Tasks Matching P1**	**# of Tasks Matching P2**	**# of Tasks Matching P3**
Node.js	398	309	223
Meteor	401	229	107
Ghost	76	77	99
Habitica	30	23	18
PDF.js	41	31	35
React	72	37	17
Serverless	81	38	23
Webpack	138	90	22
Sum	**1,237**	**834**	**544**

6.2. Metrics

We defined and used four metrics to measure a tool's capability of recommending co-changed functions: coverage, precision, recall, and F-score. We also defined the weighted average to measure a tool's overall effectiveness among all subject projects for each of the metrics mentioned above.

Coverage (Cov) is the percentage of tasks for which a tool can provide suggestion.

$$Cov = \frac{\text{\# of tasks with a tool's suggestion}}{\text{Total \# of tasks}} \times 100\% \tag{2}$$

Coverage varies within [0%, 100%]. If a tool always recommends some change(s) given a task, its coverage is 100%. All our later evaluations for precision, recall, and F-score are limited to the tasks covered by a tool. For instance, suppose that given 100 tasks, a tool can recommend changes for 10 tasks. Then the tool's coverage is $10/100 = 10\%$, and the evaluations for other metrics are based on the 10 instead of 100 tasks.

Precision (Pre) measures among all recommendations by a tool, how many of them are correct:

$$Pre = \frac{\text{\# of correct recommendations}}{\text{Total \# of recommendations by a tool}} \times 100\% \tag{3}$$

This metric evaluates how precisely a tool recommends changes. If all suggestions by a tool are contained by the ground truth, the precision is 100%.

Recall (Rec) measures among all the expected recommendations, how many of them are actually reported by a tool:

$$Rec = \frac{\text{\# of correct recommendations by a tool}}{\text{Total \# of expected recommendations}} \times 100\% \quad (4)$$

This metric assesses how effectively a tool retrieves the expected co-changed functions. Intuitively, if all expected recommendations are reported by a tool, the recall is 100%.

F-score (F1) measures the accuracy of a tool's recommendation:

$$F1 = \frac{2 \times Pre \times Rec}{Pre + Rec} \times 100\% \quad (5)$$

F-score is the harmonic mean of precision and recall. Its value varies within [0%, 100%]. The higher F-scores are desirable, as they demonstrate better trade-offs between the precision and recall rates.

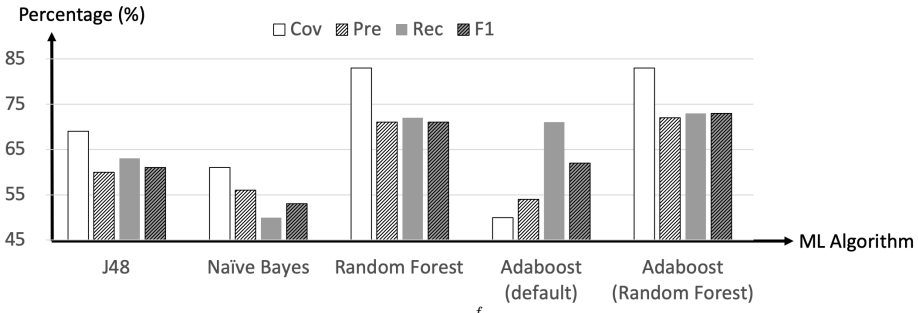
Weighted Average (WA) measures a tool's **overall effectiveness** among all experimented data in terms of coverage, precision, recall, and F-score:

$$\Gamma_{overall} = \frac{\sum_{i=1}^{8} \Gamma_i * n_i}{\sum_{i=1}^{8} n_i}. \quad (6)$$

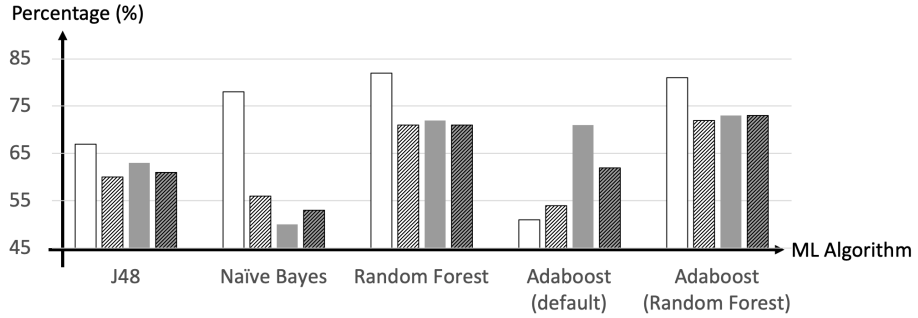
In the formula, i varies from 1 to 8, representing the 8 projects used in our evaluation (Storybook and Electron were excluded). Here, $i = 1$ corresponds to Node.js and $i = 8$ corresponds to Webpack; n_i represents the number of tasks built from the i^{th} project. Γ_i represents any measurement value of the i^{th} project for coverage, precision, recall, or F-score. By combining such measurement values of eight projects in a weighted way, we were able to assess a tool's overall effectiveness $\Gamma_{overall}$.

6.3. Sensitivity to The Adopted ML Algorithm

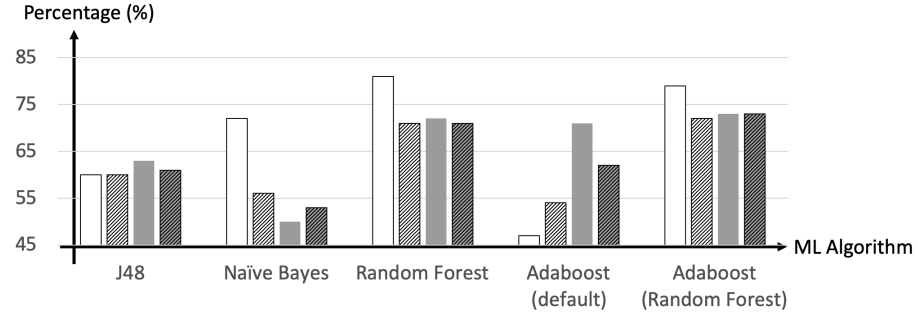
We designed CoRec to use Adaboost, with Random Forests as the weak learners to train classifiers. To make this design decision, we tentatively integrated CoRec with five alternative algorithms: J48 [42], Random Forest [43], Naïve Bayes [44], Adaboost (default), and Adaboost (Random Forest).



(a) The *CF $\xrightarrow{f}$ CF data



(b) The *CF $\xrightarrow{f}$ AF data



(c) The *CF $\xrightarrow{v}$ AV data

Figure 13: Comparison between different ML algorithms on different data sets

- **J48** builds a decision tree as a predictive model to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves).

- **Naïve Bayes** calculates the probabilities of hypotheses by applying Bayes' theorem with strong (naïve) independence assumptions between features.

- **Random Forest** is an ensemble learning method that trains a model to make predictions based on a number of different models. Random Forest trains a set of individual models in a parallel way. Each model is trained with a random subset of the data. Given a candidate in the testing set, individual models make their separate predictions and Random Forest uses the one with the majority vote as its final prediction.

- **Adaboost** is also an ensemble learning method. However, different from Random Forest, Adaboost trains a bunch of individual models (i.e., weak learners) in a sequential way. Each individual model learns from mistakes made by the previous model. We tried two variants of Adaboost: (1) Adaboost (default) with decision trees as the weak learners, and (2) Adaboost (Random Forest) with Random Forests as the weak learners.

Figure 13 illustrates the effectiveness comparison when CoRec adopts different ML algorithms. The three subfigures (Figure 13 (a)–(c)) separately present the comparison results on the data sets of *CF$\xrightarrow{f}$CF, *CF$\xrightarrow{f}$AF, and *CF$\xrightarrow{v}$AV. We observed similar phenomena in all subfigures. By comparing the first four basic ML algorithms (J48, Naïve Bayes, Random Forest, and Adaboost (default)), we noticed that Random Forest achieved the best results in all metrics. Among all datasets, Naïve Bayes obtained the lowest recall and accuracy rates. Although Adaboost obtained the second highest F-score, its coverage is the lowest probably because it uses decision trees as the default weak learners. Based on our evaluation with the first four basic algorithms, we were curious how well Adaboost performs if it integrates Random Forests as weak learners. Thus, we also experimented with a fifth algorithm: Adaboost (Random Forest).

As shown in Figure 13, Adaboost (Random Forest) and Random Forest achieved very similar effectiveness, and both of them considerably outperformed the other algorithms. But compared with Random Forest, Adaboost (Random Forest) obtained better precision, better recall, better accuracy, and equal or slightly lower coverage. Thus, we chose Adaboost (Random Forest) as the default ML algorithm used in CoRec. Our results imply that although ensemble learning methods generally outperform other ML algorithms, their effectiveness also depends on (1) what weak learners are used and (2) how we organize weak learners. Between Adaboost (Random Forest) and Adaboost (default), the only difference exists in the used weak learner (Random Forest vs. Decision Tree). Our evaluation shows that Random Forest helps improve Adaboost's performance when it is used as the weak learner. Additionally, between Random Forest and Adaboost (default), the only difference is how they combine decision trees as weak learners. Our evaluation shows that Random Forest outperforms Adaboost by training weak learners in a parallel instead of sequential way.

Finding 5: *CoRec is sensitive to the adopted ML algorithm. CoRec obtained the lowest prediction accuracy when Naïve Bayes was used, but acquired the highest accuracy when Adaboost (Random Forest) was used.*

6.4. Effectiveness Comparison with ROSE and TAR

In our evaluation, we compared CoRec with a popularly used tool ROSE [5] and a more recent tool Transitive Associate Rules (TAR) [9]. Both of these tools recommend changes by mining co-change patterns from version history. We included ROSE into our empirical comparison mainly because it has been widely cited and can be considered as the most influential existing work to recommend co-changes. We chose to also experiment with TAR because it is the state-of-the-art tool that recommends co-changes based on software history.

Specifically, ROSE mines the association rules between co-changed entities

from software version history. An exemplar mined rule is shown below:

$$\begin{array}{l}\{(\_Qdmodule.c, func, GrafObj\_getattr())\} \Rightarrow \\ \left\{\ (qdsupport.py, func, outputGetattrHook()).\ \right\}\end{array} \quad (7)$$

This rule means that whenever the function `GrafObj_getattr()` in a file _Qdmodule.c is changed, the function `outputGetattrHook()` in another file qdsupport.py should also be changed. Based on such rules, given a program commit, ROSE tentatively matches all edited entities with the antecedents of all mined rules and recommends co-changes if any tentative match succeeds. Similar to ROSE, TAR also mines association rules from version history. However, in addition to the mined rules (e.g., $E1 \Rightarrow E2$ and $E2 \Rightarrow E3$), TAR also leverages **transitive inference** to derive more rules (e.g., $E1 \Rightarrow E3$); it computes the confidence value of each derived rule based on the confidence values of original rules (e.g., $conf(E1 \Rightarrow E3) = conf(E1 \Rightarrow E2) \times conf(E2 \Rightarrow E3)$).

In our comparative experiment, we applied ROSE and TAR to the constructed prediction tasks and version history of each subject project. We configured ROSE with *support count=1* and $confidence = 0.1$, because the ROSE paper [5] mentioned this setting multiple times and it achieved the best results by balancing recall and precision. For consistency, we also configured TAR with *support count=1* and *confidence=0.1*.

As shown in Table 9, CoRec outperformed ROSE and TAR in terms of all measurements. Take Webpack as an example. Among the 138 *CF$\xrightarrow{f}$CF prediction tasks in this project, CoRec provided change recommendations for 89% of tasks; with these recommendations, CoRec achieved 71% precision, 81% recall, and 75% accuracy. On the other hand, ROSE and TAR recommended changes for only 50% of tasks; based on its recommendations, ROSE acquired only 7% precision, 29% recall, and 12% accuracy , while TAR obtained 5% precision, 34% recall, and 9% accuracy. Among the eight subject projects, the weighted average measurements of CoRec include 83% coverage, 72% precision, 73% recall, and 73% accuracy. Meanwhile, the weighted average measurements of ROSE include 53% coverage, 21% precision, 52% recall, and 29% accuracy.

Table 9: Evaluation results of CoRec, ROSE, and TAR for *CF$\xrightarrow{f}$CF tasks (%)

Project	CoRec				ROSE				TAR			
	Cov	**Pre**	**Rec**	**F1**	**Cov**	**Pre**	**Rec**	**F1**	**Cov**	**Pre**	**Rec**	**F1**
Node.js	77	68	69	69	61	24	56	34	65	15	62	24
Meteor	88	72	70	71	46	16	43	24	52	15	47	23
Ghost	73	67	74	71	50	20	53	29	50	14	57	22
Habitica	80	80	78	79	40	7	37	12	35	5	42	9
PDF.js	71	77	81	79	29	27	41	33	33	8	45	14
React	91	86	76	81	32	59	70	64	32	57	74	64
Serverless	84	77	79	78	64	20	75	32	68	16	80	27
Webpack	89	71	81	75	50	7	29	12	50	5	34	9
WA	**83**	**72**	**73**	**73**	**53**	**21**	**52**	**29**	**57**	**15**	**59**	**24**

TAR achieved 59% average recall, but its average precision and accuracy are the lowest among the three tools. Such measurement contrasts indicate that CoRec usually recommended more changes than ROSE or TAR, and CoRec's recommendations were more accurate.

In addition to *CF$\xrightarrow{f}$CF tasks, we also compared CoRec with ROSE and TAR for *CF$\xrightarrow{f}$AF and *CF$\xrightarrow{v}$AV tasks, as shown in Tables 10 and 11. Similar to what we observed in Table 9, CoRec outperformed ROSE and TAR in terms of all metrics for both types of tasks. As shown in Table 10, given *CF$\xrightarrow{f}$AF tasks, on average, CoRec achieved 81% coverage, 76% precision, 80% recall, and 78% accuracy. ROSE acquired 54% coverage, 21% precision, 48% recall, and 28% accuracy . TAR obtained 56% coverage, 16% precision, 55% recall, and 24% accuracy. In Table 11, for Serverless, CoRec achieved 70% coverage, 80% precision, 85% recall, and 82% accuracy. Meanwhile, ROSE only provided recommendations for 34% of the tasks, and none of these recommendations is correct. TAR only provided recommendations for 38% of tasks; with the recommendations, TAR achieved 1% precision, 13% recall, and 2% accuracy.

Comparing the results shown in Tables 9–11, we found the effectiveness of CoRec, ROSE, and TAR to be stable across different types of prediction tasks. Specifically among the three kinds of tasks, on average, CoRec achieved 79%–

Table 10: Result comparison among CoRec, ROSE, and TAR for *CF$\xrightarrow{f}$AF tasks (%)

Project	CoRec				ROSE				TAR			
	Cov	**Pre**	**Rec**	**F1**	**Cov**	**Pre**	**Rec**	**F1**	**Cov**	**Pre**	**Rec**	**F1**
Node.js	79	69	74	72	59	20	52	29	61	14	61	23
Meteor	86	77	82	80	40	22	44	29	46	21	50	29
Ghost	85	86	85	85	46	18	46	26	50	14	49	22
Habitica	87	77	85	81	56	4	23	7	58	2	39	4
PDF.js	65	87	88	87	22	9	28	14	23	11	58	19
React	71	84	82	83	16	66	7	13	17	67	8	14
Serverless	84	71	85	77	73	19	59	29	74	15	60	24
Webpack	75	79	85	82	53	16	46	24	56	13	49	21
WA	**81**	**76**	**80**	**78**	**54**	**21**	**48**	**28**	**56**	**16**	**55**	**24**

Table 11: Result comparison among CoRec, ROSE, and TAR for *CF$\xrightarrow{v}$AV tasks (%)

Project	CoRec				ROSE				TAR			
	Cov	**Pre**	**Rec**	**F1**	**Cov**	**Pre**	**Rec**	**F1**	**Cov**	**Pre**	**Rec**	**F1**
Node.js	79	72	77	74	55	20	65	31	56	16	74	26
Meteor	72	77	84	81	26	2	14	4	27	2	31	3
Ghost	84	75	81	78	46	18	46	26	38	8	70	14
Habitica	89	82	85	83	27	20	45	28	28	17	54	26
PDF.js	78	87	84	85	20	4	28	8	20	5	29	8
React	89	73	78	76	36	8	33	13	12	98	34	50
Serverless	70	80	85	82	34	0	0	-	38	1	13	2
Webpack	87	86	83	85	36	8	33	13	40	3	34	5
WA	**79**	**76**	**81**	**78**	**45**	**17**	**54**	**25**	**47**	**12**	**62**	**19**

83% coverage, 72%–76% precision, 73%–81% recall, and 73%–78% accuracy. On the other hand, ROSE achieved 45%–54% coverage, 17%–21% precision, 48%–54% recall, and 25%–29% accuracy; TAR achieved 47%–56% coverage, 12%–16% precision, 55%–62% recall, and 19%–24% accuracy. The consistent comparison results imply that CoRec usually recommended co-changed functions for more tasks, and CoRec's recommendations usually had higher quality.

Two major reasons can explain why CoRec worked best. First, ROSE and TAR purely use the co-changed entities in version history to recommend changes. When the history data is incomplete or some entities were never co-changed before, both tools may lack evidence to predict co-changes and thus obtain lower coverage and recall rates. Additionally, TAR derives more rules than ROSE via transitive inference. Namely, if $E1 \Rightarrow E2$ and $E2 \Rightarrow E3$, then $E1 \Rightarrow E3$. However, it is possible that $E1$ and $E3$ were never co-changed before, neither are they related to each other anyhow. Consequently, the derived rules may contribute to TAR's lower precision. Meanwhile, CoRec extracts nine features from a given commit and one feature from the version history; even though history data provides insufficient indication on the potential co-change relationship between entities, the other features can serve as supplements.

Second, ROSE and TAR observe no syntactic or semantic relationship between co-changed entities; thus, they can infer incorrect rules from co-changed but unrelated entities and achieve lower precision. In comparison, CoRec observes the syntactic relationship between co-changed entities by tracing the referencer-referencee relations; it also observes the semantic relationship by extracting features to reflect the commonality (1) between co-changed functions (*CF), and (2) between any changed function cf and the changed entity E on which cf depends (E is CF in P1, AF in P2, and AV in P3).

Although CoRec outperformed ROSE and TAR in our experiments, we consider CoRec as a complementary tool to existing tools. This is because CoRec bases its change recommendations on the three most popular RCPs we found. If some changes do not match any of the RCPs, CoRec does not recommend any change but ROSE may suggest some edits.

Table 12: The effectiveness of CoRec$_u$ when it trains and tests a unified classifier (%)

Project	Cov	Pre	Rec	F1
Node.js	72	50	57	53
Meteor	77	59	58	59
Ghost	53	61	70	65
Habitica	55	53	68	60
PDF.js	29	60	73	66
React	76	75	73	74
Serverless	54	47	61	53
Webpack	66	54	63	58
WA	**70**	**56**	**61**	**59**

Finding 6: *CoRec outperformed ROSE and TAR when predicting co-changed functions based on the three recurring change patterns (P1–P3). CoRec serves as a good complementary tool to both tools.*

6.5. Comparison with A Variant Approach

Readers may be tempted to train a unified classifier instead of three separate classifiers, because the three classifiers all take in the same format of inputs and output the same types of predictions (i.e., whether to co-change or not). However, as shown in Table 5, the commonality characteristics between co-changed functions vary with RCPs. For instance, the co-changed functions in P2 usually commonly invoke peer functions (i.e., FI), the co-changed functions in P3 often commonly read/write peer variables (i.e., VA), and the co-changed functions in P1 have weaker commonality signals for both FI and ST (i.e., common token subsequences). If we mix the co-changed functions matching different patterns to train a single classifier, it is quite likely that the extracted features between co-changed functions become less informative, and the trained classifier has poorer prediction power.

To validate our approach design, we also built a variant approach of CoRec—CoRec$_u$—that trains a unified classifier with the program commits matching either of the three RCPs (P1–P3) and predicts co-change functions with the single

classifier. To evaluate CoRec_u, we clustered the data portions matching distinct RCPs for each project, and conducted five-fold cross validation. As shown in Table 12, on average, CoRec_u recommended changes with 70% coverage, 56% precision, 61% recall, and 59% accuracy. These measured values are much lower than the weighted averages of CoRec reported in Tables 9–11. The empirical comparison corroborates our hypothesis that when data matching distinct RCPs are mixed to train a unified classifier, the classifier works less effectively.

> **Finding 7:** *CoRec_u worked less effectively than CoRec by training a unified classifier with data matching distinct RCPs. This experiment validates our approach design of training three separate classifiers.*

7. Threats to Validity

Threats to External Validity: All our observations and experiment results are limited to the software repositories we used. These observations and results may not generalize well to other JS projects, especially to the projects that use the Asynchronous Module Definition (AMD) APIs to define code modules and their dependencies. In the future, we would like to include more diverse projects into our data sets so that our findings are more representative.

Given a project P, CoRec adopts commits in P's software version history to train classifiers that can recommend co-changes for new program commits. When the version history has few commits to train classifiers, the applicability of CoRec is limited. CoRec shares such limitation with existing tools that provide project-specific change suggestions based on software version history [5, 45, 9]. To further lower CoRec's requirement to available commits in software version history, we plan to investigate more ways to extract features from commits and better capture the characteristics of co-changed functions.

Due to the time limit, we did not experiment with all commits from all subject projects. Instead, we sampled a subset of commits in each project based on the keywords "bug", "fix", "error", "adjust", and "failure" in commit messages.

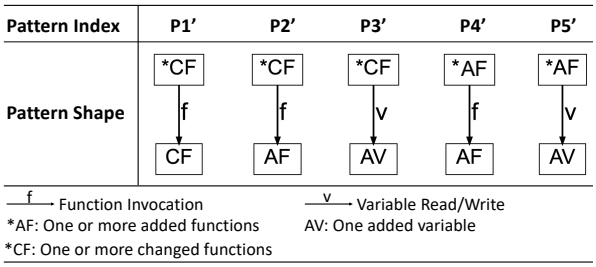


Figure 14: The five most popular recurring change patterns among all commits of Node.js

Such sampling may jeopardize the generalizability of our findings. Therefore, to validate the potential threat, we actually conducted an extra experiment to revisit the characterization study on all commits from Node.js [14]. We observed very similar results to what we reported in Section 4.

Specifically, among the 6,555 commits in Node.js that edit JS files, the majority of commits (i.e., 58%) are multi-entity edits. Within those multi-entity edits, 25% of commits involve two-entity edits, and 18% of commits are three-entity edits. The number of commits decreases as the number of edited entities increases. Our approach extracted CDGs in 62% of multi-entity edits, most of which commits (i.e., 74%) have single CDGs extracted. We extracted in total 358 RCPs from CDGs. We found that 96% of the commits with CDGs extracted have matches for RCPs. Figure 14 shows the five most popular recurring change patterns among all commits in Node.js. When comparing this figure with Figure 10, we noticed that the most popular three patterns P1–P3 remain the same across different datasets. This experiment shows that our sampling method does not considerably impact the empirical findings.

Threats to Construct Validity: When creating recommendation tasks for classifier evaluation, we always assumed that the experimented commits contain accurate information of all co-changed functions. It is possible that developers made mistakes when applying multi-entity edits. Therefore, the imperfect evaluation data set based on developers' edits may influence our empirical comparison between CoRec and ROSE. We share this limitation with prior work [5, 45, 9, 37, 46, 8, 7]. In the future, we plan to mitigate the problem

by conducting user studies with developers. By carefully examining the edits made by developers and the co-changed functions recommended by tools, we can better assess the effectiveness of different tools.

8. Related Work

The related work includes empirical studies on JS code and related program changes, JS bug detectors, and co-change recommendation systems.

8.1. Empirical Studies on JS Code and Related Program Changes

Various studies were conducted to investigate JS code and related changes [47, 48, 49, 50, 51]. For instance, Ocariza et al. conducted an empirical study of 317 bug reports from 12 bug repositories, to understand the root cause and consequence of each reported bug [47]. They observed that 65% of JS bugs were caused by the faulty interactions between JS code and Document Object Models (DOMs). Gao et al. empirically investigated the benefits of leveraging static type systems (e.g., Facebook's Flow [52] and Microsoft's TypeScript [53]) to check JS programs [49]. To do that, they manually added type annotations to buggy code and tested whether Flow and TypeScript reported an error on the buggy code. They observed that both Flow 0.30 and TypeScript 2.0 detected 15% of errors, showing great potential of finding bugs.

Silva et a. [51] extracted changed source files from software version history, and revealed six co-change patterns by mapping frequently co-changed files to their file directories. Our research is different in three ways. First, we focused on software entities with finer granularities than files; we extracted the co-change patterns among classes, functions, variables, and statement blocks. Second, since unrelated entities are sometimes accidentally co-changed in program commits, we exploited the syntactic dependencies between entities to remove such data noise and to improve the quality of identified patterns. Third, CoRec uses the identified patterns to further recommend co-changes with high quality. Wang et al. [18] recently conducted a study on multi-entity edits applied to

Java programs, which study is closely relevant to our work. Wang et al. focused on three kinds of software entities: classes, methods, and fields. They created CDGs for individual multi-entity edits, and revealed RCPs by comparing CDGs. The three most popular RCPs they found are: *CM$\xrightarrow{m}$CM (a callee method is co-changed with its caller(s)), *CM$\xrightarrow{m}$AM (a method is added, and one or more existing methods are changed to invoke the added method), and *CM$\xrightarrow{f}$AF (a field is added, and at least one existing method is changed to access the field).

Our research is inspired by Wang et al.'s work. We decided to conduct a similar study on JS programs mainly because JS is very different from Java. For instance, JS is weakly typed and has more flexible syntax rules; Java is strongly typed and variables must be declared before being used. JS is a script language and mainly used to make web pages more interactive; Java is used in more domains. We were curious whether developers' maintenance activities vary with the programming languages they use, and whether there are unique co-change patterns in JS programs. In our study, we adopted JS parsing tools, identified four kinds of entities in various ways, and did reveal some co-change patterns unique to JS programs because of the language's unique features. Surprisingly, the three most popular JS co-change patterns we observed match exactly with the Java co-change patterns mentioned above. Our study corroborates observations made by prior work. More importantly, it indicates that even though different programming languages provide distinct features, developers are likely to apply multi-entity edits in similar ways. This phenomenon sheds lights on future research directions of cross-language co-change recommendations.

8.2. JS Bug Detectors

Researchers built tools to automatically detect bugs or malicious JS code [54, 55, 56, 57, 2, 58, 59, 60]. For example, EventRacer detects harmful data races in even-driven programs [57]. JSNose combines static and dynamic analysis to detect 13 JS smells in client-side code, where smells are code patterns that can adversely influence program comprehension and software maintenance [2]. TypeDevil adopts dynamic analysis to warn developers about variables, prop-

erties, and functions that have inconsistent types [59]. DeepBugs is a learning-based approach that formulates bug detection as a binary classification problem; it is able to detect accidentally swapped function arguments, incorrect binary operators, and incorrect operands in binary operations [60]. EarlyBird conducts dynamic analysis and adopts machine learning techniques for early identification of malicious behaviors of JavaScript code [56].

Some other researchers developed tools to suggest bug fixes or code refactorings [61, 62, 63, 64, 65, 66, 67]. With more details, Vejovis suggests program repairs for DOM-related JS bugs based on two common fix patterns: parameter replacements and DOM element validations [64]. Monperrus and Maia built a JS debugger to help resolve "crowd bugs" (i.e., unexpected and incorrect outputs or behaviors resulting from the common and intuitive usage of APIs) [65]. Given a crowd bug, the debugger sends a code query to a server and retrieves all StackOverflow answers potentially related to the bug fix. An and Tilevich built a JS refactoring tool to facilitate JS debugging and program repair [67]. Given a distributed JS application, the tool first converts the program to a semantically equivalent centralized version by gluing together the client and server parts. After developers fixed bugs in the centralized version, the tool generates fixes for the original distributed version accordingly. In Model-Driven Engineering, ReVision repairs incorrectly updated models by (1) extracting change patterns from version history, and (2) matching incorrect updates against those patterns to suggest repair operations [68].

Our methodology is most relevant to the approach design of ReVision. However, our research is different in three aspects. First, our research focuses on entity-level co-change patterns in JS programs, while ReVision checks for consistencies different UML artifacts (e.g., the signature of a message in a sequence diagram must correspond to a method signature in the related class diagram). Second, the co-changed recommendation by CoRec intends to complete an applied multi-entity edit, while the repair operations proposed by ReVision tries to complete consistency-preserving edit operations. Third, we conducted a large-scale empirical study to characterize multi-entity edits and experimented CoRec

with eight open-source projects, while ReVision was not empirically evaluated.

8.3. Co-Change Recommendation Systems

Approaches were introduced to mine software version history and to extract co-change patterns [51, 69, 70, 36, 5, 71, 45, 9, 72, 73, 74, 75, 76, 6]. Specifically, Some researchers developed tools (e.g., ROSE) to mine the association rules between co-changed entities and to suggest possible changes accordingly [5, 71, 45, 9, 72, 76, 6]. Some other researchers built hybrid approaches by combining information retrieval (IR) with association rule mining [73, 74, 75]. Given a software entity E, these approaches use IR techniques to (1) extract terms from E and any other entity and (2) rank those entities based on their term-usage overlap with E. Meanwhile, these tools also apply association rule mining to commit history in order to rank entities based on the co-change frequency. In this way, if an entity G has significant term-usage overlap with E and has been co-changed a lot with E, then G is recommended to be co-changed with E.

Shirabad et al. developed a learning-based approach that predicts whether two given files should be changed together or not [36]. In particular, the researchers extracted features from software repository to represent the relationship between each pair of files, adopted those features of file pairs to train an ML model, and leveraged the model to predict whether any two files are relevant (i.e., should be co-changed) or not. CoRec is closely related to Shirabad et al.'s work. However, it is different in two aspects. First, CoRec predicts co-changed functions instead of co-changed files. With finer-granularity recommendations, CoRec can help developers to better validate suggested changes and to edit code more easily. Second, our feature engineering for CoRec is based on the quantitative analysis of frequent change patterns and qualitative analysis of the commonality between co-changed functions, while the feature engineering by Shirabad is mainly based on their intuitions. Consequently, most of our features are about the code commonality or co-evolution relationship between functions; while the features defined by Shirabad et al. mainly focus on file names/paths, routines referenced by each file, and the code comments together

with problem reports related to each file.

Wang et al. built CMSuggester—an automatic approach to suggest method-level co-changes in Java programs [7, 8]. CoRec and CMSuggester are different in four aspects. First, the tools take in different inputs. To recommend co-changes for a project's commit, CoRec requires two inputs: the commit and the history data of that project. CMSuggester generates suggestions purely based on the given commit. Second, the tools implement different methodologies. CoRec is a data-driven instead of rule-based approach; it requires for co-change data to train an ML model while CMSuggester requires tool builders to hardcode the suggestion strategies. Third, the target programming languages are different. CoRec targets JS, so it has unique handlings for ASTs of JS programs to parse four kinds of entities: classes, functions, variables, and blocks. CMSuggester targets Java, so it has simpler processing for ASTs of Java programs to parse three kinds of entities: classes, methods, and fields. Fourth, the tools have different applicable scopes. CoRec can recommend changes based on three RCPs: *CF$\xrightarrow{f}$CF, *CF$\xrightarrow{f}$AF, and *CF$\xrightarrow{v}$AV; CMSuggester only recommends changes based on the last two patterns mentioned above. Overall, CoRec is more flexible due to its usage of ML and is applicable to more types of co-change scenarios.

9. Conclusion

It is usually tedious and error-prone to develop and maintain JS code. To facilitate program comprehension and software debugging, we conducted an empirical study on multi-entity edits in JS projects and built an ML-based co-change recommendation tool CoRec. Our empirical study explored the frequency and composition of multi-entity edits in JS programs, and investigated the syntactic and semantic relevance between frequently co-changed entities. In particular, we observed that (i) JS software developers frequently apply multi-entity edits while the co-changed entities are usually syntactically related; (ii) there are three most popular RCPs that commonly exist in all studied JS code repositories: *CF$\xrightarrow{f}$CF, *CF$\xrightarrow{f}$AF, and *CF$\xrightarrow{v}$AV; and (iii) among the entities

matching these three RCPs, co-changed functions usually share certain commonality (e.g., common function invocations and common token subsequences).

Based on our study, we developed CoRec, which tool extracts code features from the multi-entity edits that match any of the three RCPs, and trains an ML model with the extracted features to specially characterize relationship between co-changed functions. Given a new program commit or a set of entity changes that developers apply, the trained model extracts features from the program revision and recommends changes to complement applied edits as necessary. Our evaluation shows that CoRec recommended changes with high accuracy and outperformed two existing techniques. In the future, we will investigate novel approaches to provide finer-grained code change suggestions and automate test case generation for suggested changes.

10. Acknowledgements

We thank anonymous reviewers for their valuable comments on our earlier version of the paper. This work was supported by NSF-1845446, and National Key R&D Program of China No. 2018YFC083050.

References

[1] The 10 most popular programming languages, according to the Microsoft-owned GitHub, `https://www.businessinsider.com/most-popular-programming-languages-github-2019-11` (2019).

[2] A. M. Fard, A. Mesbah, Jsnose: Detecting javascript code smells, in: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 116–125.

[3] A. Saboury, P. Musavi, F. Khomh, G. Antoniol, An empirical study of code smells in javascript projects, 2017, pp. 294–305. `doi:10.1109/SANER.2017.7884630`.

[4] R. Ferguson, Introduction to JavaScript, Apress, Berkeley, CA, 2019, pp. 1–10. doi:10.1007/978-1-4842-4395-4_1.
URL https://doi.org/10.1007/978-1-4842-4395-4_1

[5] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, Mining version histories to guide software changes, in: Proc. ICSE, 2004, pp. 563–572.

[6] T. Rolfsnes, L. Moonen, S. D. Alesio, R. Behjati, D. Binkley, Aggregating association rules to improve change recommendation, Empirical Software Engineering 23 (2) (2018) 987–1035.

[7] Y. Wang, N. Meng, H. Zhong, Cmsuggester: Method change suggestion to complement multi-entity edits, in: Proc. SATE, 2018, pp. 137–153.

[8] Z. Jiang, Y. Wang, H. Zhong, N. Meng, Automatic method change suggestion to complement multi-entity edits, Journal of Systems and Software 159.
URL http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-85073505497&site=eds-live&scope=site

[9] M. A. Islam, M. M. Islam, M. Mondal, B. Roy, C. K. Roy, K. A. Schneider, [research paper] detecting evolutionary coupling using transitive association rules, in: Proc. SCAM, 2018, pp. 113–122.

[10] Z. P. Fry, W. Weimer, A human study of fault localization accuracy, in: 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–10. doi:10.1109/ICSM.2010.5609691.

[11] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, T. N. Nguyen, Recurring bug fixes in object-oriented programs, in: ACM/IEEE International Conference on Software Engineering, 2010, pp. 315–324. doi:http://doi.acm.org/10.1145/1806799.1806847.

[12] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, L. Bairavasundaram, How do fixes become bugs?, in: Proc. ESEC/FSE, 2011, pp. 26–36.

[13] J. Park, M. Kim, B. Ray, D.-H. Bae, An empirical study of supplementary bug fixes, in: IEEE Working Conference on Mining Software Repositories, 2012, pp. 40–49.

[14] Nodejs node, https://github.com/nodejs/node (2020).

[15] Fs: make callback mandatory to all async functions, https://github.com/nodejs/node/commit/21b0a27 (2016).

[16] What Is ES6 and What Javascript Programmers Need to Know, https://www.makeuseof.com/tag/es6-javascript-programmers-need-know/ (2017).

[17] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, Chianti: A tool for change impact analysis of java programs, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '04, ACM, New York, NY, USA, 2004, pp. 432–448. doi:10.1145/1028976.1029012.
URL http://doi.acm.org/10.1145/1028976.1029012

[18] Y. Wang, N. Meng, H. Zhong, An empirical study of multi-entity changes in real bug fixes, in: Proc. ICSME, 2018, pp. 287–298.

[19] Esprima (2020).
URL https://esprima.org/

[20] typed-ast-util, https://github.com/returntocorp/typed-ast-util (2020).

[21] Fix some fibers vs SQLite issues, https://github.com/meteor/meteor/commit/e9a88b00b9cdd35eb281c7113fcaa5155f006ea3 (2020).

[22] Meteor, https://github.com/meteor/meteor (2020).

[23] J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras,

Sweden - September 15 - 19, 2014, 2014, pp. 313–324. doi:10.1145/2642937.2642982.
URL http://doi.acm.org/10.1145/2642937.2642982

[24] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub)graph isomorphism algorithm for matching large graphs, IEEE Transactions on Pattern Analysis and Machine Intelligence 26 (10) (2004) 1367–1372. doi:10.1109/TPAMI.2004.75.

[25] Ghost, https://github.com/TryGhost/Ghost (2020).

[26] Habitrpg habitica, https://github.com/HabitRPG/habitica (2020).

[27] Mozilla pdf, https://github.com/mozilla/pdf.js/ (2020).

[28] Facebook react, https://github.com/facebook/react (2020).

[29] Serverless, https://github.com/serverless/serverless (2020).

[30] Webpack, https://github.com/webpack/webpack (2020).

[31] Storybook, https://github.com/storybookjs/storybook (2020).

[32] Electron, https://github.com/electron/electron (2020).

[33] Http2: introducing HTTP/2, https://github.com/nodejs/node/commit/e71e71b5138c3dfee080f4215dd957dc7a6cbdaf (2017).

[34] J. Ingeno, Software Architect's Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts, Packt Publishing, 2018.

[35] F. McCarey, M. Ó. Cinnéide, N. Kushmerick, Rascal: A recommender agent for agile reuse, Artificial Intelligence Review 24 (3) (2005) 253–276. doi:10.1007/s10462-005-9012-8.
URL https://doi.org/10.1007/s10462-005-9012-8

[36] J. S. Shirabad, T. C. Lethbridge, S. Matwin, Mining the maintenance history of a legacy software system, in: Proc. ICSM, 2003, pp. 95–104.

[37] N. Meng, M. Kim, K. McKinley, Lase: Locating and applying systematic edits, in: Proc. ICSE, 2013, pp. 502–511.

[38] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670. doi:10.1109/TSE.2002.1019480.

[39] C. K. Roy, J. R. Cordy, Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: 2008 16th IEEE International Conference on Program Comprehension, 2008, pp. 172–181. doi:10.1109/ICPC.2008.41.

[40] L. Li, H. Feng, W. Zhuang, N. Meng, B. Ryder, Cclearner: A deep learning-based clone detection approach, 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2017) 249–260.

[41] Y. Freund, R. E. Schapire, Experiments with a new boosting algorithm, in: Proceedings of the Thirteenth International Conference on International Conference on Machine Learning, ICML'96, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996, pp. 148–156.

[42] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[43] A. Liaw, M. Wiener, Classification and regression by randomforest, R News 2 (3) (2002) 18–22.
URL https://CRAN.R-project.org/doc/Rnews/

[44] D. D. Lewis, Naive (bayes) at forty: The independence assumption in information retrieval, in: C. Nédellec, C. Rouveirol (Eds.), Machine Learning: ECML-98, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 4–15.

[45] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, D. W. Binkley, Generalizing the analysis of evolutionary coupling for software change impact analysis, in: Proc. SANER, 2016, pp. 201–212.

[46] Tan, Ming, Online defect prediction for imbalanced data, Master's thesis, University of Waterloo (2015).

[47] F. Ocariza, K. Bajaj, K. Pattabiraman, A. Mesbah, An empirical study of client-side javascript bugs., 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on, Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on (2013) 55 – 64.
URL http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.6681338&site=eds-live&scope=site

[48] M. Selakovic, M. Pradel, Performance issues and optimizations in javascript: An empirical study, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 61–72. doi:10.1145/2884781.2884829.

[49] Z. Gao, C. Bird, E. T. Barr, To type or not to type: Quantifying detectable bugs in javascript, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 758–769. doi:10.1109/ICSE.2017.75.

[50] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, A. Mesbah, Bugsjs: a benchmark of javascript bugs, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 90–101. doi:10.1109/ICST.2019.00019.

[51] L. L. Silva, M. T. Valente, M. A. Maia, Co-change patterns: A large scale empirical study, Journal of Systems and Software 152 (2019) 196 – 214. doi:https://doi.org/10.1016/j.jss.2019.03.014.
URL http://www.sciencedirect.com/science/article/pii/S0164121219300597

[52] Flow: A Static Type Checker for JavaScript, https://flow.org (2020).

[53] TypeScript - JavaScript that scales., https://www.typescriptlang.org (2020).

[54] M. Cova, C. Kruegel, G. Vigna, Detection and analysis of drive-by-download attacks and malicious javascript code, in: Proceedings of the 19th International Conference on World Wide Web, WWW '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 281–290. doi:10.1145/1772690.1772720.
URL https://doi.org/10.1145/1772690.1772720

[55] F. S. Ocariza Jr., K. Pattabiraman, A. Mesbah, Autoflox: An automatic fault localizer for client-side javascript, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 31–40.

[56] K. Schütt, M. Kloft, A. Bikadorov, K. Rieck, Early detection of malicious behavior in javascript code, in: Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence, AISec '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 15–24. doi:10.1145/2381896.2381901.
URL https://doi.org/10.1145/2381896.2381901

[57] V. Raychev, M. Vechev, M. Sridharan, Effective race detection for event-driven programs, ACM SIGPLAN NOTICES 48 (10) (2013) 151 – 166.
URL http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edswsc&AN=000327697300008&site=eds-live&scope=site

[58] J. Park, Javascript api misuse detection by using typescript, in: Proceedings of the Companion Publication of the 13th International Conference on Modularity, MODULARITY '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 11–12. doi:10.1145/2584469.2584472.
URL https://doi.org/10.1145/2584469.2584472

[59] M. Pradel, P. Schuh, K. Sen, Typedevil: Dynamic type inconsistency analysis for javascript, 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (2015) 314.
URL http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=110064267&site=eds-live&scope=site

[60] M. Pradel, K. Sen, Deepbugs: A learning approach to name-based bug detection, Proc. ACM Program. Lang. 2 (OOPSLA). doi:10.1145/3276517.
URL https://doi.org/10.1145/3276517

[61] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, F. Tip, Tool-supported refactoring for javascript, SIGPLAN Not. 46 (10) (2011) 119–138. doi:10.1145/2076021.2048078.
URL https://doi.org/10.1145/2076021.2048078

[62] F. Meawad, G. Richards, F. Morandat, J. Vitek, Eval begone! semi-automated removal of eval from javascript programs, SIGPLAN Not. 47 (10) (2012) 607–620. doi:10.1145/2398857.2384660.
URL https://doi.org/10.1145/2398857.2384660

[63] S. H. Jensen, P. A. Jonsson, A. Møller, Remedying the eval that men do, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, Association for Computing Machinery, New York, NY, USA, 2012, pp. 34–44. doi:10.1145/2338965.2336758.
URL https://doi.org/10.1145/2338965.2336758

[64] F. Ocariza, K. Pattabiraman, A. Mesbah, Vejovis: Suggesting fixes for javascript faults, in: Proceedings - International Conference on Software Engineering, no. 1, Electrical and Computer Engineering, University of British Columbia, 2014, pp. 837–847.
URL http://login.ezproxy.lib.vt.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-84993660437&site=eds-live&scope=site

[65] M. Monperrus, A. Maia, Debugging with the Crowd: a Debug Recommendation System based on Stackoverflow, Research Report hal-00987395, Université Lille 1 - Sciences et Technologies (2014).
URL https://hal.archives-ouvertes.fr/hal-00987395

[66] M. Selakovic, M. Pradel, Poster: Automatically fixing real-world javascript performance bugs, 2015 ICSE International Conference on Software Engineering. (2015) 811.
URL http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=111750044&site=eds-live&scope=site.

[67] K. An, E. Tilevich, Catch & release: An approach to debugging distributed full-stack javascript applications, in: M. Bakaev, F. Frasincar, I.-Y. Ko (Eds.), Web Engineering, Springer International Publishing, Cham, 2019, pp. 459–473.

[68] M. Ohrndorf, C. Pietsch, T. Kehrer, ReVision: A tool for history-based model repair recommendations, in: Proc. ICSE-Companion, 2018, p. 105.

[69] H. Gall, K. Hajek, M. Jazayeri, Detection of logical coupling based on product release history, in: Proc. ICSM, 1998, pp. 190–198.

[70] H. Gall, M. Jazayeri, J. Krajewski, CVS release history data for detecting logical couplings, in: Proc. IWPSE, 2003, pp. 13–23.

[71] A. T. T. Ying, G. C. Murphy, R. T. Ng, M. Chu-Carroll, Predicting source code changes by mining change history., IEEE Trans. Software Eng. 30 (9) (2004) 574–586.

[72] H. Kagdi, J. I. Maletic, B. Sharif, Mining software repositories for traceability links, in: Proc. ICPC, 2007, pp. 145–154.

[73] H. H. Kagdi, M. Gethers, D. Poshyvanyk, Integrating conceptual and logical couplings for change impact analysis in software, Empirical Software Engineering 18 (2012) 933–969.

[74] M. Gethers, B. Dit, H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: Proc. ICSE, 2012, pp. 430–440.

[75] M. B. Zanjani, G. Swartzendruber, H. Kagdi, Impact analysis of change requests on source code based on interaction and commit histories, in: Proc. MSR, 2014, pp. 162–171.

[76] L. L. Silva, M. T. Valente, M. de Almeida Maia, Co-change clusters: Extraction and application on assessing software modularity, Trans. Aspect-Oriented Software Development 12 (2015) 96–131.