Client Insourcing: Bringing Ops In-House for Seamless Re-engineering of Full-Stack JavaScript Applications

Kijin An
Software Innovation Lab, Virginia Tech
ankijin@vt.edu

Eli Tilevich
Software Innovation Lab, Virginia Tech
tilevich@cs.vt.edu

ABSTRACT

Modern web applications are distributed across a browser-based client and a cloud-based server. Distribution provides access to remote resources, accessed over the web and shared by clients. Much of the complexity of inspecting and evolving web applications lies in their distributed nature. Also, the majority of mature program analysis and transformation tools works only with centralized software. Inspired by business process re-engineering, in which remote operations can be insourced back in house to restructure and outsource anew, we bring an analogous approach to the re-engineering of web applications. Our target domain are full-stack JavaScript applications that implement both the client and server code in this language. Our approach is enabled by Client Insourcing, a novel automatic refactoring that creates a semantically equivalent centralized version of a distributed application. This centralized version is then inspected, modified, and redistributed to meet new requirements. After describing the design and implementation of Client Insourcing, we demonstrate its utility and value in addressing changes in security, reliability, and performance requirements. By reducing the complexity of the non-trivial program inspection and evolution tasks performed to meet these requirements, our approach can become a helpful aid in the re-engineering of web applications in this domain.

CCS CONCEPTS

• **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*.

KEYWORDS

Software Engineering, Re-Engineering, Web Applications, JavaScript, Mobile Apps, Program Analysis & Transformation, Middleware

ACM Reference Format:
Kijin An and Eli Tilevich. 2020. Client Insourcing: Bringing Ops In-House for Seamless Re-engineering of Full-Stack JavaScript Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Developers often need to re-engineer web applications to address requirement changes made only after deployment and usage. Re-engineering captures evolutionary modifications that range from maintenance tasks to architecture-level changes [4]. A re-engineering effort can involve adding a major feature, protecting against security vulnerabilities, or removing performance bottlenecks. Modifying existing web applications requires complex program analysis and modification operations that are hard to perform and even harder to verify. One of the main causes of this complexity is the distributed execution model of web applications.

In this model, a web application's execution flows across the separate address spaces of its client and server parts. All remote interactions are typically implemented by means of middleware libraries. As a result, the control flow of web applications can be highly complex, with their business and communication logic intermingled. That complexity hinders all tracing and debugging tasks. In addition, distributed execution over the network makes web applications vulnerable to partial failure and non-determinism.

Program analysis is central to software comprehension. The web is predominated by dynamic languages, which defeat static analysis techniques. Hence, to comprehend programs written in dynamic languages, such as JavaScript, requires dynamic analysis. Software debugging hinges on the ability to repeat executions deterministically [22, 27]. However, many web applications are stateful, with certain client server interactions changing the server's state. It can be quite laborious and error-prone to restore the original state to be able to repeat a remote buggy operation [14, 21, 30]. All in all, it is the presence of both distribution and stateful execution that makes it so hard to trace and modify web applications.

In this paper, we draw inspiration from business process re-engineering that can bring remote operations in-house via *insourcing*. Once the insourced operations are redesigned and restructured, some of them can be outsourced anew. As argued above, the notion of local operations being easier to analyze and restructure than remote ones equally applies to web applications.

Specifically, the approach presented herein first automatically transforms a web application, comprising a client communicating with a remote server, to run as a centralized program. The resulting centralized variant retains to a large degree the semantics of the original application, but replaces all remote operations with local ones. The centralized variant becomes easier to analyze and modify not only because it has no remote operations, but also because the majority of program analysis and transformation approaches and tools have been developed for centralized programs. After the centralized variant is modified to address the new requirements and the modifications have been verified, it is then redistributed again into a re-engineered distributed web application. Our target domain are web applications written entirely in JavaScript, both the client and server parts; such applications are referred to as *full-stack JavaScript applications*. We take advantage of the monolingual nature of such applications to streamline our implementation.

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In a web application, clients communicate with the server by means of the HTTP protocol, typically in a request/response pattern. However, from the implementation perspective, the HTTP functionality can be supported by a variety of middleware libraries with vastly dissimilar APIs [25]. To be able to identify and replace the HTTP communication functionality, a web application may need to be executed multiple times under different inputs. However, some remote interactions cause the server to change its state. For example, a client can pass a parameter to the server, which would store that parameter in the server-side database. In addition, the non-database state can change as well (e.g., adding the parameter to the JavaScript list of displayed items). In different states, the server may respond dissimilarly, thus making it impossible to identify HTTP middleware API calls, so they can be correctly replaced with corresponding local calls of the insourced functionalities.

The focal point of our approach is Client Insourcing, a new automatic refactoring that undoes distribution by gluing the local and remote parts of a distributed application together. Our approach can precisely identify the functionality of HTTP middleware—irrespective of its API and in the presence of stateful operations—by combining program instrumentation, profiling, and fuzzing in a novel way. Our ideas are realized in our reference implementation—Java Script Remote Client Insourcing (JS-RCI). We evaluate our approach's value, correctness, and utility by applying JS-RCI to re-engineer a set of real-world web applications.

The contribution of this paper is three-fold:

(1) We introduce a technique that identifies the HTTP middleware functions used to send and receive HTTP commands in a full-stack JavaScript application. This technique eliminates the need to specialize our approach for the multitude of HTTP middleware libraries and their APIs.
(2) We create *Client Insourcing*—a novel automatic refactoring that creates a semantically equivalent centralized version of a distributed application by integrating remote functionalities with local code and replacing middleware communication with direct function calls. This refactoring moves to the client not only the server's business logic implemented in JavaScript, but also the referenced database functionality, including the relational database schema in SQL.
(3) We evaluate the wide applicability of Client Insourcing in re-engineering real-world full-stack JavaScript applications. Specifically, we apply our approach to re-engineer 10 subject distributed applications, both two-tier and three-tier (including the database), to meet new security, reliability, and performance requirements.

The rest of this paper is structured as follows. Section 2 motivates and summarizes our approach. Sections 3 and 4 present the design and implementation specifics of the Client Insourcing refactoring, respectively. Section 5 reports on how we applied Client Insourcing to streamline three representative re-engineering scenarios of web applications. Section 6 discusses various applicability issues pertaining to our approach. Section 7 compares our approach with the related state of the art. Section 8 outlines future work directions and presents concluding remarks.

2 RE-ENGINEERING WEB APPLICATIONS

Developers often find themselves having to re-engineer an actively used application to ensure its continued utility, reliability, and safety. When interacting with an application in real-world settings, users may discover and report inefficiencies and imperfections. Users may request that new features be added to an application to increase its utility. As users discover existing faults and request new features, developers can decide to re-engineer the application to deliver an improved version to the users. Re-engineering modifications can range from routine maintenance and evolution tasks to major architectural transformations. Next, we demonstrate two examples of re-engineering full-stack JavaScript applications.

2.1 Example Apps

The code snippets in Figure 1 come from two third-party full-stack JavaScript applications *realty-rest*[1] (left) and *recipebook*[2] (right), with both of their client and server parts shown. Both applications rely on the network for their client and server parts to communicate with each other. The primary user base of *realty-rest* are real-estate brokers, licensed professionals that sell and purchase various properties on behalf of their clients. Due to the nature of their business operations, real-estate brokers lead highly mobile professional lives, moving from location to location to show properties to potential buyers. Hence, as a mobile app, *realty-rest* is well-aligned with the needs of its users, who rely on the app to be readily available, responsive, and reliable. To start using the app, a user selects a property from the list of all properties registered with the system. The selected property can then be updated or deleted, with the app's client then sending HTTP commands to the server, (e.g., `DELETE /property/favorite` to remove a property from the list of favorites, etc). The HTTP commands are wrapped into distribution middleware (angular2/http) with JavaScript API. Specifically, the client invokes `HTTP.delete` passing a URL parameter, with angular2/http delivering the invocation to the server and calling function `unfavorite` there. This function finds and deletes the passed `property`, returning the updated list of favorites to the client. angular2/http marshals both `property` and the result-to-return as JSON-encoded messages. The client unmarshals the returned result to update the GUI. The *recipebook* maintains a list of cooking recipes at the server, so different clients could retrieve and update the maintained recipes. *recipebook* uses a different middleware library to wrap its HTTP commands—angularJS, whose JavaScript API differs from that of angular2/http. While *realty-rest* is a two-tier app (JavaScript client and sever), *recipebook* is three-tier (adding a database tier).

Next, we present examples of how *realty-rest* and *recipebook* may need to be re-engineered to address new requirements.

2.2 Adapting to Disconnected Operation

Examining the history of *realty-rest* reveals that some of this app's functionalities have been moved between its client and server sites[3]. Since scant documentation makes it hard to ascertain the reason for these moves, we next discuss a typical new feature that enables distributed apps to continue operation in the absence of a network

[1] realty-rest (https://github.com/ccoenraets/ionic2-realty-rest)
[2] recipebook (https://github.com/9bitStudios/recipebook)
[3] ionic2-realty (https://github.com/ccoenraets/ionic2-realty)

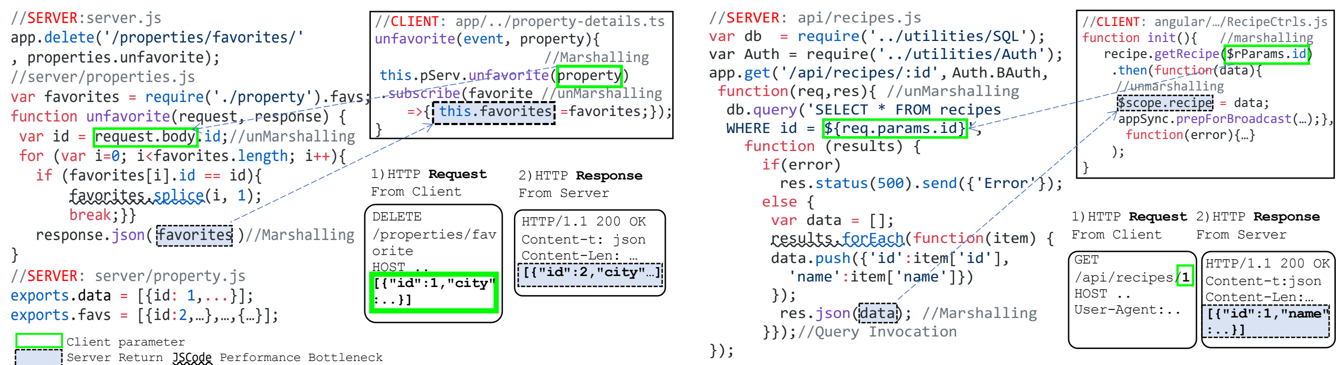


Figure 1: Motivating Distributed Apps *realty-rest/recipebook* **and highlighted Client-/Server-side code**

connection. In particular, if users need to operate a mobile app in locations with limited or intermittent network connectivity, the app has to deliver its core business functionality without relying on any remote services. To enable such offline operations, several strategies have been proposed [28]. One such strategy is *replication*, which replicates a remote component locally, so the local copy acts as a proxy of its remote counterpart. A consistency protocol keeps both copies in sync. A naïve strategy for replicating a remote functionality would be just to copy its complete source files to the client, adapting the copied code by hand as necessary. However, such complete copying unnecessary replicates functionalities, some of which become "dead code."

2.3 Enhancing Privacy

Enterprises often find themselves in need to enhance user privacy in a released application. Consider a request to keep the *realty-rest* user's property browsing histories private from other real-estate brokers due to business competition reasons. To ensure user privacy, certain server-side functionalities (e.g., Customer Relationship Management (CRM)) can be redistributed to a special server that requires authentication before giving access to sensitive information. In fact, *realty-rest* indeed has gone through a similar modification, as evidenced by the existence of *realty-salesforce*[4], which provides the same business functionality, but takes advantage of third-party trusted identification and security features. To re-engineer *realty-rest* into *realty-salesforce*, programmers would have to identify and migrate the relevant functionality to another server, modifying the client to communicate with different servers (regular and secure).

2.4 Improving Performance

If a substantial subset of users becomes unsatisfied with application performance, programmers may be asked to identify and remove performance bottlenecks. The left side of Figure 1 displays the server function `unfavorite`, which contains a known performance bottleneck, rooted in the usage of `favorite.splice(i,1)`, an inefficient API for removing collection items. In fact, an actual pull request[5] states that `Array.splice()`'s performance is between 1.5 and 10 times slower than that of a customized implementation, comprising a `for` loop iteration and `Array.pop()`. To be able to identify this particular source of the experienced performance bottleneck, programmers either would have to be intimately familiar with the peculiarities of JavaScript APIs or to rely on detailed execution profiling, typically available only for centralized programs. *recipebook* also contains a similarly inefficient `forEach` loop[6]. Notice that the distributed control flow that invokes these inefficient functions, starting from the graphical actions at the client, traversing the network through layers and layers of middleware, and finally executing the functions at the server. The invocation flows can be interrupted by network volatility and authentication failures. Hence, it is both complex control flows and possible failures that make it hard to isolate the performance of a web application's function.

2.5 Client Insourcing to the Rescue

Next, we explain how Client Insourcing can facilitate the re-engi - neering tasks outlined above.

Redistribution Client Insourcing creates a redistributable centralized variant devoid of the unnecessary middleware functionality. Once the variant is modified, it can be redistributed automatically. Numerous complementary research efforts have focused on automating the process of distributing centralized applications, with automatic transformation tools released to the public [17, 19, 48]. Because the majority of existing refactoring techniques are designed for regular centralized applications, they can be applied at will to centralized variants. For example, the *Extract Function* refactoring can be used to separate some privacy-sensitive code within a function into a separate function to be executed in a different environment. After the sensitive code portions are separated into their own encapsulation units, the resulting program can be redistributed, placing the sensitive units to execute in separate privacy-enforcing server environments.

Isolated Profiling What if business logic can be precisely isolated from middleware and distribution-related functionality? Then the isolated code can be easily profiled to ascertain its performance characteristics and identify any performance bottlenecks. Client Insourcing enables such isolated profiling by removing middleware and gluing the remote parts of a web application together.

Offline Operation Client Insourcing can enable offline operation, without copying any unnecessary code from the server to the client,

[4] realty-salesforce (https://github.com/ccoenraets/ionic2-realty-salesforce)
[5] Perfective Modification for Array.splice() (https://github.com/nodejs/node/pull/20453)
[6] A modification request to remove this inefficiency appears here: https://github.com/elastic/apm-agent-nodejs/pull/1275

by replicating only the remote functionality's subset needed at the client. The replicated subset can include both JavaScript code and data persisted in a database.

3 DESIGN & REFERENCE IMPLEMENTATION

In this section, we explain our design options and then detail the specifics of our implementation of the Client Insourcing refactoring.

3.1 Design Overview

We give an overview of the main design decisions behind Client Insourcing via specific examples. Consider the task of moving the server functionalities of `DEL /favorite` or `GET /recipe/:id` to execute at the clients (Figure 1). Instead of invoking these functions via middleware that handles communication, partial failures, and authentication, they would become regular local functions to be called directly. Hence, all middleware-based code would have to be replaced with direct function calls.

Consider the service `DEL /favorite`, whose business logic is encapsulated within the server-side `unfavorite` function. We want to insource `unfavorite` so it can be called as a regular local function. However, we cannot simply move this function from the server to the client, as its business logic and middleware functionality are intermingled. In addition, the `exports.favorite` array, referenced in the body of `unfavorite`, is declared externally. If `unfavorite` and `exports.favorite` are not moved together, invoking the function locally would raise an error. Hence, we must move all the referenced externally declared program elements to the client as well. JS-RCI identifies the exact boundaries of the server functionality to insource. However, some dependent business logic of `GET /recipe/:id` is not confined to JavaScript code only. JS-RCI also transparently insources code that persists data in a relational database.

```
//app/../property-details.js
import {j5ga2} from './j5ga2';
unfavorite { const IS_SYNC = false;
  if (IS_SYNC) {//synchronous call
    this.favorites = j5ga2(property.id);
    return;
  } //default: non-blocking call
  new Promise((resolve,reject) => {
    var out_j5ga2 = j5ga2(property.id);
    resolve(out_j5ga2);
  }).then(res => this.favorites = res);
}
```

```
//app/../b8f9a.js
exports.favs = [{id: 1,city:'B,..}];
//app/../j5ga2.js
var favorites=require('./b8f9a').favs;
export function j5ga2(input){
 var tmpv1 = input;
 var id = tmpv1;
 for (var i=0; i< favorites.length;
i++){… favorites.splice(i, 1);…}
 tmpv0 = favorites;
 var output = tmpv0;
 return output;}//extracted function
```

Figure 2: Transformed and generated code to insource a functionality `DELETE /properties/favorite` in *realty-rest* app

3.2 Identifying the Code to Insource

Next, we present our solution for automating the steps above, realized as the *Client Insourcing Refactoring*. One of our design goals was to make sure that this domain-specific refactoring is not too burdensome for the programmer. We assume that the refactored applications come with a set of standard test cases, and that the application of these cases is automated. It is during the application of such test cases, when JS-RCI detects the marshalling/unmarshalling points of the functionality to insource at the client invocations. Intuitively, the purpose of detecting these marshalling/unmarshalling points in the client code is to identify the entry/exit execution points of the remote functionality to insource. These points correspond to the locations in the client code, at which remote invocation parameters are marshalled to be transferred across the network, and the remote invocation's results are unmarshalled to be used in the subsequent client execution.

To extract all the server code of the remote functionality to insource, JS-RCI uses symbolic execution. We assume that the server is implemented in Node.js and define the execution rules as pertaining to this framework's architectural conventions. First of all, JS-RCI normalizes server code to facilitate to detect entry/exit execution points and extract the executed JavaScript code. To that end, JS-RCI additionally introduce temporal local variables and makes JavaScript Statement to have a single operation (i.e., `tmpv0` and `tmpv1` in Figure 2). For symbolic execution, we use z3 [7], parameterized with our own set of rules and facts. For example, the profiled parameters and return results of a remote functionality are added as new z3 facts. Figure 4 shows the overall process of Client Insourcing.

3.3 Exploiting Asynchrony

Notice that in a distributed client-server application, the remotely invoked functionalities running at the server, and the client code invoking these functionalities, run in separate address spaces that are not shared (unless the application runs on top of some distributed shared memory system [32], which is not a standard option for web applications). The parameters passed to remote invocations and the invocation results are copied between the client and the server heaps, always creating a new copy rather than mutating any existing program state. Hence, in a distributed application that uses application-layer middleware (e.g., HTTPClient), the client and the server parts share *no mutable state* (See Figure 3). Following this observation, one can conclude that the client and the server parts have *no non-middleware dependencies* between them. That is, in such distributed applications, the only way for the client code to invoke a server-side functionality is by making a remote invocation via middleware. To maintain this semantics, our design also provides a single entry point to invoke the insourced functionality, a function previously invoked via a middleware API call at the server. It is these insights that make it possible to safely execute the insourced code asynchronously, without any need for synchronization! Our design of Client Insourcing takes advantage of these insights by executing the insourced functionality *asynchronously by default*. In particular, the generated code makes use of the `Promise` framework that exposes asynchronous execution via a standardized interface that uses the programming idioms congruent with the design of JavaScript.

For a specific example, consider the code listing in Figure 2 that shows the generated client code for `DEL /favorite`. Notice that the default invocation model for this insourced function is asynchronous, a runtime behavior that is put into effect by creating a new instance of a `Promise` closure. Once the asynchronous execution of `j5ga2` completes, the *Promise* framework invokes the callback `resolve` to handle the successful execution. Since our design aims for versatility, we provide an option for the insourced functionality to be invoked synchronously as a regular blocking local call. This behavior can be put into effect by setting the value of the `boolean` variable `IS_SYNC` to **`true`**.

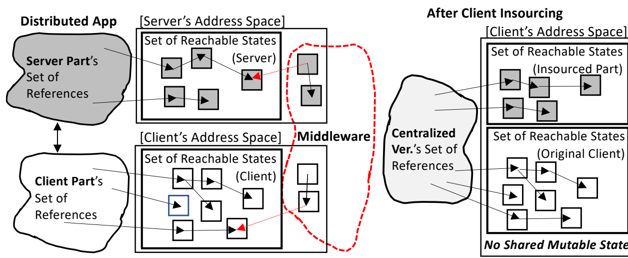


Figure 3: Reachable States between Server and Client parts

4 IMPLEMENTATION SPECIFICS

In this section, we provide some additional details pertaining to our implementation choices.

4.1 Detecting Marshalling Points in Client/Server Program

In a full-stack JavaScript application, the client interacts with the server in the request/response pattern, exchanging data in JSON or XML formats. Client Insourcing determines which middleware API calls send and receive the HTTP protocol commands through the following automatic and application-agnostic procedure.

First, the round-trip traffic of the client/server interactions is recorded. Then, JS-RCI parses the request/response data to obtain the deserialized values of *client parameters* and *server return*. To that end, JS-RCI uses the GoReplay [13] tool to capture live network traffic, not only to record/replay the HTTP interactions, but also to extract the used HTTP commands. To capture business logic (as compared to fault handling logic), JS-RCI only processes the responses with the status code of 400 (i.e., successful execution).

Next, JS-RCI replays the recorded round-trip execution that invokes the remote functionality to insource. Both the client and server parts are dynamically instrumented to keep track of values for (1) arguments and returns of the function invocations (2) reading and writing variables. JS-RCI keeps comparing the values of the invocations and variables to identify the ones equal to *client parameter* and *server return*. To instrument the invocations and variable accesses, JS-RCI uses the Jalangi2 callback APIs [40].

To identify the entry points at the server, JS-RCI keeps comparing the values for recorded *client parameter* of the remote functionality. That is, the parameter has been unmarshalled and is about to be used. To identify the exit point at the server, JS-RCI follows a similar procedure, but looks for the value recorded as the *server return* of executing the remote functionality. Finding an equal value read or written determines the exit point of the remote functionality. That is, the return value is about to be marshalled and sent across the network to the client. One may wonder: how does our approach determine that the equality comparison indeed identifies the entry and exit points of the remote functionality rather than some intermediate values that also happen to be equal to the values of *client parameter* and *server return*? To identify the entry and exit points at the server, our analysis identifies the first instance of the *client parameter* equality and the last instance of the *server return* value equality. Unlike its server-side logic, the analysis identifies the last instance of the *client parameter* equality and the first instance of the *server return* value equality.

4.1.1 Fuzzing Request/Response Messages.

Even with these arrangements in place, it is still possible to misidentify the correct entry and exit points, particularly if the parameters or return results are primitive types, such as built-in numbers or strings (i.e., 0 or 1 values of id in `findById` service). To prevent such misidentification, JS-RCI populates the original round-trip content by padding the HTTP header and body data with random bits. A fuzzing dictionary is also applied to fuzzable primitives types: string has the possible values "JSRCIStr" and integer has the possible values from "9,000" and to "1,0000." For instance, JS-RCI encodes "1" as "9,001". For a service without a client parameter (i.e., `findAll` type services), JS-RCI fuzzes the request with "JSRCIStr" so JS-RCI can locate the function block's begin as the entry point.

4.1.2 Achieving the Idempotency for Record/Replay Executions.

Despite the stateless nature of the RESTful architecture that guides the design of WWW, few realistic web applications are truly stateless. In fact, every HTTP request can change the server's state. These changes hinder the precision of our detection of the server's marshalling points, introducing *false-negatives*. Even HTTP traffic were replied with identical requests, a stateful server is likely to behave differently in 1) marshalling its response output or 2) entering the remote functionality through a different point (e.g., if a visited entry is deleted, it cannot be revisited).

Testing web applications deterministically requires that test cases be isolated [14, 29]. Otherwise, the same test case can yield dissimilar results when executed with the same input. Restoring the server to its original state by hand would be expensive in realistic web applications, requiring a manual reset of the relevant database tables and a fresh restart of the server. In contrast, JS-RCI fully automates the process to achieve the idempotent execution of all HTTP requests. To maintain the original server's state, JS-RCI interleaves an automatically generated *restore* operation, run between all successive record or replay executions. Similarly to a prior approach that checkpoints PHP web application ([14]), JS-RCI initiates the restore operation with a special HTTP request. Similarly to manipulating fuzzed request messages, JS-RCI generates the *restore* operations by enhancing original HTTP requests with the new "JSRCIRestore" parameter. To be able to restore the server state, JS-RCI first saves the initial values of all server's global variables, so they can be restored on demand. Also, as part of its restore operation, JS-RCI executes transaction control operations between every SQL invocations, so the database rollbacks to its previous state.

As its specific implementation strategy, JS-RCI uses jalangi2, whose *shadow execution* instruments the original JavaScript code, so the server events can be hooked dynamically. First, JS-RCI detects all (1) post declarations of global variables ( ) and (2) pre/post *Call Expressions* of SQL statements (f). Then, it uses two customized shadow executions at (1), $^{0} = store^{1}\ ^{\circ}$ to serialize and store the state of all global variables and $restore^{1}\ ,\ ^{0\circ}$ to reset all global variables to their original values, hooked by *restore* HTTP commands. To restore the database state, JS-RCI uses shadow execution $in\ oke^{1} f, sql_stat^{\circ}$, which invokes *Call Expression* of a SQL statement f with a new SQL clause as the argument. $in\ oke^{1} f$,"Start TRANSACTION"$^{\circ}$ and $in\ oke^{1} f$,"ROLLBACK"$^{\circ}$ are executed at pre

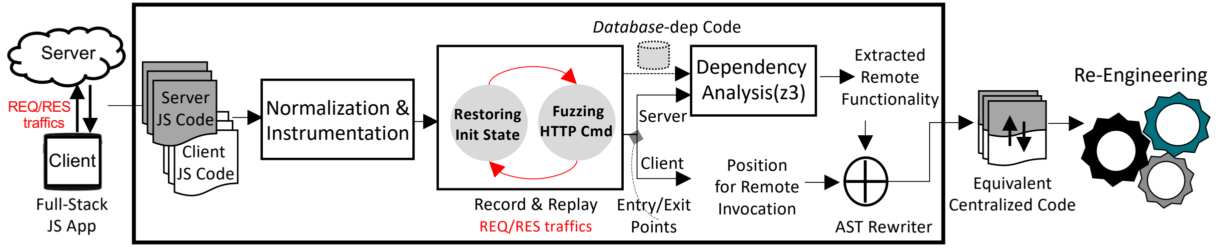


Figure 4: Overall process for Client Insourcing

and post invocations of f, respectively. JS-RCI executes these operations only once for the nested SQL invocations.

4.2 Identifying the Relevant Server Code

One of the factors that complicates the *Client Insourcing* refactoring is that the code comprising the functionality of the insourced functionality may not be confined to the boundaries of a single function or even the same script. While the entry point of the remote execution can be a JavaScript function, this function can be invoking other functions or reference variables declared elsewhere. When insourcing a remote functionality, all this dependent code must be moved together to the client to create a self-sufficient local call that no longer relies on any server-based code.

To determine the data dependencies between the entry/exit points of a distributed application's remote functionality, we draw lessons provided by the state-of-the-art JavaScript analysis frameworks [15, 16, 44]. JSdep [44] logically hypotheses a DATA-DEP relation between JavaScript statements based on read/write facts, a point-to-analysis model of GateKeeper [15] and a control flow analysis [16] . For instance, an assignment statement ASSIGN becomes a fact that implies READ and WRITE relations for the variables involved. READ and WRITE on the same variable between different statements imply a DATA-DEP relation at the statement level. We

$$
\begin{aligned}
\text{ASSIGN}(stmt_1, \ _1, \ _2) &\quad \text{//var } \ _1 = \ _2; \quad \text{ is variable, } stmt \text{ is statement} \\
\text{WRITE}(stmt_1, \ _1) &\quad \text{ASSIGN}(stmt_1 \ \ _1, \ _2) \\
\text{READ}(stmt, \ _2) &\quad \text{ASSIGN}(stmt, \ _1, \ _2) \\
\text{DATA-DEP}(stmt_1, stmt_2) &\quad \text{READ}(stmt_1, \ _1) \wedge \text{WRITE}(stmt_2, \ _1) \\
\ldots &
\end{aligned}
$$

extend JSdep's knowledge base with the rules and facts, necessary to model the execution of middleware-based statements. In particular, we define the UNMARSHAL/MARSHAL rules to identify the entry and exit points, whose WRITE/READ clauses are inferred from the logged profiling data. To that end, JS-RCI encodes the REF facts by using the logged values to symbolically copy the unmarshalled/-marshalled values ($V^{u_{id}}_{unMar}/V^{u_{id}}_{Mar}$, u_{id} is an unique execution id such as "J5ga2") into the local variables as follows:

$$
\begin{aligned}
&\text{//the entry point at the server} \\
&\text{UNMARSHAL}(stmt_1, \ _{unMar}, u_{id}) \\
\text{WRITE}(stmt_1, \ _{unMar}) \wedge\ &\text{REF}(\ _{unMar}, V^{u_{id}}_{unMar}) \\
&\text{//the exit point at the server} \\
&\text{MARSHAL}(stmt_1, \ _{Mar}, u_{id}) \\
\text{READ}(stmt_1, \ _{Mar}, u_{id}) \wedge\ &\text{REF}(\ _{Mar}, V^{u_{id}}_{Mar})
\end{aligned}
$$

Based on the resulting knowledge base, JS-RCI can query the executed statements $stmt_n$ for the presence of variable u_{id}. Predicate EXECUTEDSTMTS is a conjunction of two clauses: the first clause expresses the dependent statements for the parameter marshalling statement, while the second clause expresses the dependent statements for the result unmarshalling statement, both specific to the server execution. Because the DATA-DEP relation is transitive, one can obtain the executed statements from the entry/exit points, as expressed by the following set operations:

$$
\begin{aligned}
&\text{EXECUTEDSTMTS}(stmt_n, u_{id}) \\
(\text{DATA-DEP}(stmt_n, stmt_1) \wedge\ &\text{MARSHAL}(stmt_1, \ _1, u_{id})) \wedge \\
(\colon \text{DATA-DEP}(stmt_n, stmt_2) \wedge\ &\text{UNMARSHAL}(stmt_2, \ _2, u_{id}))
\end{aligned}
$$

4.3 Insourcing Database-Dependent Code

Our approach can also insource code that persists data in a relational database. To that end, we take advantage of the ubiquity of SQL. Recall that JS-RCI dynamically instruments string values used as arguments and return values in all function calls. To identify the entry point for database-related operations, JS-RCI examines the function calls whose strings arguments represent the CRUD operations (Create, Read, Update, and Delete). Consider the code snippet in Figure 1. JS-RCI detects that the following *Call Expression* is a READ operation, as it is a SQL SELECT statement:

```
db.query("SELECT * FROM recipes WHERE id=id", function(result)..);
```

Although the server and the client are written in JavaScript and their respective database engines accept the same SQL statements, the JavaScript APIs of these engines differ. So it would be impossible to simply move this *Call Expression* and its dependent statements (e.g., `var db = require('../utilities/SQL');`) to the client. Hence, JS-RCI adapts the server-side database API to that of the client rather than copying the database-specific statements verbatim. With these API calls translated, developers can simply migrate the server-side data schema and tables. Notice that database engines store their data in dissimilar proprietary formats.

As a specific example, consider how JS-RCI translates the database API calls of MySQL[7] to those of *alasql*[8]. By extracting the arguments and return values of function calls, JS-RCI extracts table names and their columns, thereby inferring a complete data schema of the insourced code. Extracting the actual table content requires a different approach, as the WHERE clause and numerical

[7] https://github.com/mysqljs/mysql
[8] https://github.com/agershun/alasql

functions, such as COUNT, return only a subset of table rows. To retrieve all database data, JS-RCI instruments the server code by using the shadow execution *in oke*[1]`db.query`, "SELECT * FROM recipes"), which is introduced in Section 4.1.2. To infer the database schema from the extracted entries, JS-RCI uses tableschema-py[9]. Finally, JS-RCI uses the CREATE and INSERT commands with *alasql* to create tables and insert the extracted data into them, respectively, for the client-side database.

5 EVALUATION

To determine how feasible and useful our approach is, we conduct an empirical evaluation driven by the following questions:

RQ1. Effort Saved by Client Insourcing : How much programmer effort is saved by applying JS-RCI ? We measure the saved effort as the number of lines of code that would need to be copied and modified by hand. JS-RCI saves this effort automating these manual source code changes. (Section 5.2)

RQ2. Correctness of Client Insourcing : Does Client Insourcing preserve the business logic of full-stack JavaScript applications? Are existing standard use-cases still applicable to the centralized variants of the subject applications? (Section 5.3)

RQ3. Value for Adaptive Tasks : How much redundant code can Client Insourcing eliminate by *replicating* only the necessary remote functionality? Are our centralized variants amenable to be *redistributed* with a third-party automated distribution tool? (Section 5.4)

RQ4. Value for Perfective Tasks : How suitable are the centralized variants of distributed subjects for isolating and removing common performance bottlenecks? How much does Client Insourcing reduce the task complexity as compared to the original debugging process? (Section 5.5)

5.1 Evaluation Setup

To evaluate our approach, we have applied it to insource **61** different remote executions of **10** full-stack JavaScript applications [3, 6, 8, 9, 26, 34–36, 42, 45]. Table 1 summarizes the information about invoking these remote functionalities for each application. These remote services differ in their HTTP methods (e.g., GET, POST, PUT etc.), types of parameters, return results, and business logic.

To confirm that our approach is widely applicable, we selected as our evaluation subjects open-source full-stack JavaScript applications with dissimilar HTTP frameworks used to implement their client (Tier 1), server (Tier 2) and database (Tier 3) parts: **Tier1:** JQuery, Ajax, fetch, axios, AngularJS, and Angular2-TS; **Tier2:** Express, koa.js, and Restify, and **Tier3:** MySql, Postgres, and knex.js.

5.2 Saving Effort with Client Insourcing

Although developers can insource remote components by hand, the resulting program transformations can quickly become laborious and error-prone, especially for functionalities scattered across multiple script files and database-dependent code appearing in non-JavaScript files. Hence, the value of JS-RCI lies in automating the

Table 1: Subject Distributed Apps and Client Insourcing Results

Subject Apps (tier1,tier2,tier3)	**Remote Services**		*C&P•M* (ULOC)
recipebook (AngularJS $ Express $ MySQL)	GET	/recipes	22/45
	GET/PUT/POST/DEL	/recipes:id	72/172
	POST	/ingredients	25/48
	GET/PUT/DEL	/ingredients:id	74/207
	POST	/directions	26/57
	GET/PUT/DEL	/directions:id	60/130
DonutShop (Ajax $ Express $ knex)	GET/POST	/donuts	22/88
	GET/POST/DEL	/donuts:id	29/155
	GET/POST	/employee	20/71
	GET/POST/DEL	/employee:id	29/138
	GET/POST	/shops	16/83
	GET/DEL	/shops:id	19/128
res-postgresql (axios $ restify $ Postgres)	GET/POST	/user	22/71
	GET/PUT/DEL	/user	40/120
med-chem-rules (fetch $ koa.js $ knex)	GET	/hbone	9971/9994
	GET	/molecular	9974/9997
theBrownNode (JQuery $ Express)	GET	/users/search	37/65
	GET	/users/search/id	36/64
Bookworm (AngularJS $ Express)	GET	/api/ladywithpet	394/409
	GET	/api/thedea	394/409
	GET	/api/theredroom	394/409
	GET	/api/thegift	394/409
	GET	/api/wallpaper	394/409
	GET	/api/offshore	394/409
	GET	/api/bigtripup	394/409
	GET	/api/amontillado	394/409
realty_rest (Angular2 $ Express)	GET	/properties	284/297
	GET	/properties:id	287/300
	GET	/brokers	86/99
	GET	/brokers:id	90/103
	GET/POST/DEL	/prprts/favorites	34/73
	POST	/properties/likes	291/304
ConferenceApp (Angular2 $ Express)	GET	/findAllSpeakers	13/66
	GET	/findSpeakerById	15/68
	GET	/findAllSessions	43/117
	GET	/findSessionById	46/119
Employee Dir (Angular2 $ Express)	GET	/employees	22/44
	GET	/employees/id	38/60
shopping-cart (Angular2 $ Express)	GET/POST/DEL	/cart-items	79/130
total	**61**		**24.9K/26.6K**

transformations required to insource these components. With JS-RCI completely automating the refactoring, the programmer would not have to modify any code by hand. To estimate the effort saved by JS-RCI, we use the ULOC (Uncommented Lines of Code) that would have to be copied at the server and pasted to the client as well as the ULOC that would have to be modified at the client for each remote service. Thus, modified client code (**M**) includes the copied/pasted code (**C&P**). For the 61 remote services of 10 applications, JS-RCI eliminates the need to modify the client code as many as **26,685** ULOCs in total, **20,073** ULOCs are database code.

5.3 Correctness of Client Insourcing

The applicability of JS-RCI hinges on whether Client Insourcing preserves the execution semantics (i.e., business logic) of the refactored applications, a property we refer to as *correctness.* A subject application's original and refactored versions are expected to successfully pass the same test cases. Some of the tests that come with our subjects are also distributed, invoking server-side functionalities through HTTP middleware. To use their remote parameters and results as test invariants, we manually transformed these tests for local execution without middleware. Altogether we ran 61 test cases against the original and insourced versions of our subject applications, with all of them successfully passing. It is possible that for some complex or esoteric cases, the correctness of Client

[9] https://github.com/frictionlessdata/tableschema-py

Insourcing would not be as stellar, but by examining why a test case failed, the programmer can always correct the insourced code.

5.3.1 The E ectiveness and Correctness of Detecting the Marshalling Points. Recall that in Section 4, we proposed two search strategies—*Idempotent Execution* and *Fuzzing*—to detect the marshalling points of a refactored application. To compare and contrast the effectiveness and correctness of these strategies, we ran our analysis procedure with each of these strategies in isolation.

We observed that Idempotent Execution with its Record/Replay phases removes the *false-negatives* in the detected marshalling points for stateful servers. Our results show that subject applications with only safe (or read-only) operations are not affected by the restoring process (20/61). However, we discovered that idempotent execution is critical for the majority of our subjects (41/61). Specifically, having been changed by HTTP PUT/POST/DELETE requests, global variables were restored correctly in realty-rest and database entries were restored in other subjects.

In contrast, Fuzzing removes *false-positives* for detecting marshalling points. We discovered that Fuzzing proved effective also in twelve cases of our subjects (12/61). Hence, to infer the correct set of marshalling points, while removing both false-negatives and positives, JS-RCI applies both strategies in turn.

Table 2: Correctness affected by Search Strategies

subject	State -less	Data -Base	all	w/o Fuzzing	w/o Idem_Ex
theBrownNode	✓	✗	2/2	0/2	2/2
Bookworm	✓	✗	8/8	0/8	8/8
ConferenceApp	✓	✗	4/4	4/4	4/4
EmployeeDir	✓	✗	2/2	2/2	2/2
shopping-cart	✗	✗	3/3	3/3	0/3
realty-rest	✗	✗	8/8	6/8	2/8
recipebook	✗	✓	13/13	13/13	0/13
DonutShop	✗	✓	14/14	14/14	0/14
res-postgresql	✗	✓	5/5	5/5	0/5
med-chem-rules	✓	✓	2/2	2/2	2/2
Total			**100%(61/61)**	**80%(49/61)**	**32%(20/61)**

5.4 Insourcing's Value for Adaptive Tasks

5.4.1 Value of Automated Enabling of Disconnected Operation. In lieu of Client Insourcing, developers would have to replicate remote functionalities by hand. Unassisted by program analysis, a programmer remains unaware which specific code entities comprise a remote functionality that needs to be replicated. Hence, a safe option for manually replicating any non-trivial remote functionality would be to first duplicate the entire server-side source file at the client, and then adapt the duplicated code as necessary. Notice that such copy-and-modify procedures invariably introduce some unnecessary code, which is never used but still needs to be deployed and maintained. Hence, in our evaluation, we count the number of lines of such unnecessary code that could result from copying the entire source file from the server to the client.

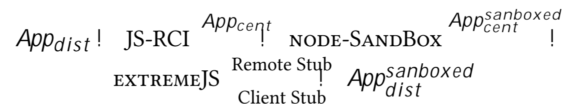
To identify the code portions that are indeed unnecessary to replicate the remote functionalities under consideration, we first count the total lines of JavaScript code taken to implement the original server parts of each subject app (S_{LOC}). To replicate all remote functionalities, programmers would copy S_{LOC} to the client and

Table 3: Replication

Subject Apps	S_{LOC}	S^{CI}_{LOC}	S_{LOC} - S^{CI}_{LOC} (Unnecessary LOC)
theBrownNode	120	76	44
Bookworm	340	299	41
realty-rest	457	420	37
ConferenceApp	78	51	27
EmployeeDir	56	35	21
shopping-cart	48	26	22
recipebook	624	376	126
DonutShop	455	308	147
res-postgresql	73	28	45
med-chem-rule	10228	9976	252

adapt them as necessary. The copied S_{LOC} are intermingled with various unnecessary parts, including middleware, fault handling, or no-longer relevant comments. The values of S_{LOC} are computed by examining the programmer-written files and their dependencies deplofyed in the Node.js server. In contrast, Client Insourcing extracts from the server only the lines of code required to implement the replication disconnected operation (S^{CI}_{LOC}). For simplicity, we assume that the entire remote functionality is replicated for each subject application. To estimate the number of lines of code that Client Insourcing saves from being replicated unnecessarily, we subtract S_{LOC} from S^{CI}_{LOC} as shown in Table 3.

5.4.2 Value of Centralized Variants for Redistribution. Client insourcing creates a redistributable (centralized) application variant that can be *refactored* and *enhanced* using any state-of-the-practice program transformation tools and then *distributed anew* using any state-of-the-art ditribution tools. We applied two JavaScript refactoring tools on our centralized variants: Node-SandBox[10] for security enhancements and extremeJS [47] for redistribution. Node-SandBox prevents untrusted JavaScript code from executing infinite loops or consuming large volumes of heap memory in the isolated code. However, sanboxing frameworks incur a heavy performance penalty on the isolated code, and as such must be used sparingly, if the application is to remain usable. Hence, the code to sandbox is typically isolated from the rest of the application to run in its own process and address space. extremeJS automatically distributes centralized JavaScript applications at the function level of granularity.



In our evaluation, we measure the additional execution time incurred by sandboxing only a subset of the remote functionality vs. the entire original remote functionality. This comparison highlights the importance of isolating only the code that needs to be sandboxed. Figure 5 shows by how much sandboxing increases the execution time for two versions of the subject applications: (1) only the needed subset of the server part is isolated (*SandBox_part*); (2) the entire server part is isolated (*SandBox_all*). The observed differences in execution time between these two versions are quite striking, clearly showing that sandboxing the entire server part is impractical.

[10]Node-SandBox (https://github.com/patriksimek/vm2)

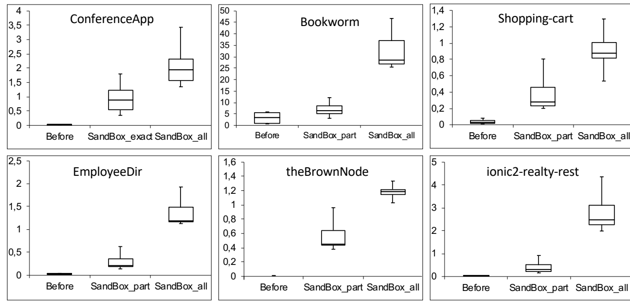


Figure 5: Redistribution with Sandboxing

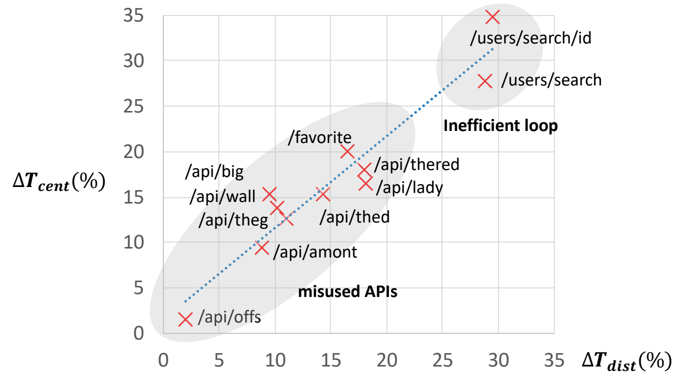


Figure 6: Scatter Plot and Regression Test for ΔT_{dist} versus ΔT_{cent}

5.5 Insourcing's Value for Perfective Tasks

Consider the problem of identifying the source of a performance inefficiency or bottleneck in a distributed application. First, one has to be able to exclude the reasons of misconfiguration or network volatility among the potential causes. Then, one has to make sure the application is free of known architectural anti-patterns [37]. For example, consecutive fine-grained remote invocations can be batched to take advantage of better progress being made in increasing the bandwidth as compared to the latency characteristics of modern networks [31]. However, the sources of inefficiency can be more subtle than those stemming from ill-conceived architectural decisions. At some point, the debugging focus may need to switch to the programmer-written code. The JavaScript ecosystem features numerous frameworks and libraries, so the same functionality can be implemented in a variety of ways, each of which may have its own performance characteristics. Choosing one programming idiom over another can have a dramatic effect on the overall application performance [11, 12]. Given the divergent performance characteristics of different JavaScript APIs, several prior work directions have focused on identifying and removing common sources of inefficiency. The approach presented in [39] empirically identifies recurring patterns of inefficient program performance, so they can be restructured, thereby improving the overall performance. That kind of restrucuring is a common example of perfective modifications. However, the majority of the state-of-the-art approaches that identify and remove performance inefficiencies target centralized programs. Client Insourcing can make these approaches applicable to distributed applications.

We applied the approach presented in [39] to the centralized variants of our subject applications produced by means of Client Insourcing. Out of 61 subjects, 11 ended up containing some known patterns of performance inefficiency. For example, *Bookworm* repetitively misused unoptimized string API patterns: `data.split("...").join("asdf").split(".").join("asdf")`. By taking the network and middleware functionality out of the list of suspected causes of performance problems, Client Insourcing enables so-called "isolated profiling," which isolates the programmer-written code to be used as the sole target of analysis and optimization efforts. To demonstrate the value of Client Insourcing, we removed all the pointed-out 11 performance bottlenecks from both the original subjects and their centralized variants. As it turns out, the bottleneck removals improved the performance of both versions (distributed and centralized) of each subject. Figure 6 summarizes the observed performance improvements. For the original distributed subjects (ΔT_{dist}), the improvements range between 29.5% and 2.0%. For their centralized variants (ΔT_{cent}), the improvements range between 34.8% and 1.6%. We also applied a linear regression analysis to compute how closely ΔT_{dist} and ΔT_{cent} correlate with each other, resulting in $\Delta T_{cent} = \mathbf{1.0089} \quad \Delta T_{dist} + \mathbf{1.556}$. This equation shows that ΔT_{cent} and ΔT_{dist} are almost perfectly correlated, so centralized variants can indeed serve as reliable and convenient proxies for an important class of performance debugging and optimization tasks.

In addition, Client Insourcing reduces the complexity of the debugging process by streamlining the debugged subject's execution flow: from the complexity of distributed execution over the Web to the simplicity of centralized execution. To quantify the actual value of debugging the centralized variant of a web application instead of its original distributed version, we compared the total execution time taken by invoking distributed functionalities vs. their local insourced counterparts. We assumed that the debugging task was identifying performance bottlenecks, so we heavily instrumented our benchmarks before measuring their execution performance. As it turns out, insourcing reduces a distributed functionality's execution time by more than **90%** on average. Given that debugging typically involves repeated executions, having much faster subjects to debug should improve the efficiency of the debugging process.

5.6 Threats to Validity

The validity of our evaluation results is subject to both internal and external threats that we discuss in turn next.

Internal Threats. One of our evaluation criteria is the performance of the JavaScript code generated by our implementation of the Client Insourcing refactoring. The performance of JavaScript code is known to be heavily affected by specific design and implementation choices. Similarly, our own JavaScript coding practices are likely to have affected the observed performance characteristics. For example, rather than directly inject the insourced code segments into the client source files, we choose to create brand new source files for each insourced languages declaration, with the new files simply included in the original files. Client Insourcing could have

been implemented in a variety of other ways, possibly yielding different software engineering and performance metrics.

External Threats. All our performance measurements were performed on (Dell-OptiPlex5050, running the JavaScript V8 Engine(v 6.11.2). Due to the popularity of JavaScript, the issue of maximizing the efficiency of JavaScript engines has come to the forefront of system design[39]. Although V8 is a state-of-the-art JavaScript engine, it has its competitors, such as SpiderMonkey. Hence, the absolute performance of our experiments could differ if our measurements were run in a different execution environment.

6 DISCUSSION

Our approach works only with relational databases interfaced with by means of SQL queries. Some non-SQL databases, such as MongoDB, use a distinct syntax in its client API. It should be possible to support the dissimilar CRUD operations of non-SQL databases, and we plan to explore such support as a future work direction.

For various reasons, some remote functionalities cannot be insourced to run on the client, thus making it impossible to create a centralized variant of certain distributed applications. In those cases in which distribution is inevitable, some application resources, naturally remote to the rest of the functionality, cannot change their locality. For instance, news readers display the stories deposited to some centralized repository. It would be impossible to move the news functionality away from the repository to the client, without manually creating some mock components that realistically emulate the appearance of news content locally. In other words, some remote functionalities may depend on resources that cannot be easily migrated away from their host environment for reasons that include relying on server-specific APIs or being dependent on some hard-to-move infrastructure components.

In addition to standard commands, HTTP also provides a separate WebSocket interface that opens a dedicated TCP/UDP connection after a round-trip handshake. WebSocket-based communication is fundamentally asynchronous and is used mostly in streaming scenarios. Although Client Insourcing can also help in the re-engineering of web applications that use WebSocket for non-streaming scenarios, we left the support for this part of HTTP as a future work direction.

Some web applications may span across more than two tiers. Our reference implementation assumes a two-tier client-server application with a possible server-side SQL database, in which both tiers are implemented in JavaScript. It should be possible to extend Client Insourcing to multi-tier applications, perhaps by applying the two-tier technique pairwise to each respective pair of tiers. At the same time, flattening tiers may not work well for mobile execution environments, which are known to be resource-scarce.

7 RELATED WORK

Several prior approaches conquer the complexity introduced by middleware functionality through abstraction and modeling techniques. A dynamic analysis platform analyzes full-stack JavaScript applications by abstracting away middleware communication, so it can be emulated in dynamic profiling scenarios [5]. [1] studies implicit relations between asynchronous and event-driven entities that are spread over the client and server sides of a distributed execution. JS-RCI is unique in its ability to remove the no-longer-necessary middleware functionality and compute the server-side dependent source code, which may not even be declared in the same source file as the insourced functionality's entry point.

Several recent techniques automatically integrate portions of a program's source in another program with systems such as CodeCarbonReply [43] and Scalpel [2] supporting this functionality for C/C++ programs. However, these works studied how to integrate two independent centralized programs.

Our reference implementation of Client Insourcing, JS-RCI, relates to advanced program analysis techniques for JavaScript, due to its target domain—cross-platform mobile applications. The JavaScript language constructs for programming event-based applications that communicate asynchronously (i.e., *callback*, *promises*) have been statically analyzed via formal reasoning based on a calculus [23, 24]. Existing dynamic analysis tools [20, 40] are known to scale poorly to handle whole JavaScript program analysis. In dynamic symbolic execution (DSE), a program is symbolically executed in place of concrete input values [38]. MultiSE[41] effectively generates testing input values of a JavaScript program by using a value summary in Jalangi2 to speed-up dynamic symbolic execution.

JS-RCI is related to re-engineering tools that automatically transform applications [10, 33, 48]. Guided by scalable zero-knowledge proofs, the ZØ compiler[10] preserves user privacy by splitting existing code into distributed multi-tier applications. Cloud offloading [48] improves the energy efficiency of a mobile app by splitting it into the client and server parts.

JavaScript debugging is an active research area. BLeak [46] and MemInsight [18] identify memory leaks by checking for sustained memory growth patterns between consecutive executions. JSweeter [49] detects performance bottlenecks caused by JavaScript type mutations. However, these tools work only with centralized JavaScript applications that are run on a single V8 engine.

8 FUTURE WORK AND CONCLUSION

We designed and implemented our approach with the assumption that it would be applied to monolingual execution environments, such as that of full-stack JavaScript applications. However, many modern distributed applications are multilingual, with the client and server parts written in different languages, often quite dissimilar. It might be possible to extend Client Insourcing to such multilingual environments by supplementing our design with automatic cross-language translation. In other words, to extend the applicability of Client Insourcing to multilingual distributed applications, one can build upon our design and reference implementation by adding a cross-language translation component to the last phase of the refactoring process. This new component would automatically translate the insourced code from the server language to that of the client. When invoking the insourced translated code, the differences between the calling conventions would have to be reconciled.

We have presented an approach that facilitates the profiling, adaptation, and securing of full-stack JavaScript applications. The approach is enabled by Client Insourcing, a novel automated refactoring that integrates remote functionalities with local code, thereby creating a semantically equivalent centralized variant of a distributed application. We showed how this centralized version can be

analysed and modified more easily than its distributed counterpart, to be then redistributed automatically with all the modifications in place. The pervasiveness of distribution highlights the need for novel automated techniques for the re-engineering of web applications, and Client Insourcing can potentially become a useful building block for such techniques.

ACKNOWLEDGMENTS

This research is supported by the NSF through the grants # 1650540 and 1717065.

REFERENCES

[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding Asynchronous Interactions in Full-stack JavaScript. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1169–1180. https://doi.org/10.1145/2884781.2884864

[2] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 257–269. https://doi.org/10.1145/2771783.2771796

[3] Bookworm. 2019. https://github.com/davidwoodsandersen/Bookworm. (2019).

[4] Eric J Byrne. 1992. A conceptual foundation for software re-engineering. In *Proceedings of the Conference on Software Maintenance*. IEEE, 226–235.

[5] Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2018. Orchestrating Dynamic Analyses of Distributed Processes for Full-stack JavaScript Programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2018)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/3278122.3278135

[6] ConferenceApp. 2019. https://github.com/tkssharma/Ionic-conferenceApp. (2019).

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] Donuts. 2019. https://github.com/VinniiOtchkov/Donuts. (2019).

[9] EmployeeDir. 2019. https://github.com/ccoenraets/employee-directory-services. (2019).

[10] Matthew Fredrikson and Benjamin Livshits. 2014. ZØ: An Optimizing Distributing Zero-Knowledge Compiler. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 909–924. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/fredrikson

[11] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 357–368.

[12] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. 94–105.

[13] GoReplay. 2018. (2018). https://github.com/buger/goreplay.

[14] Marco Guarnieri, Petar Tsankov, Tristan Buchs, Mohammad Torabi Dashti, and David Basin. 2017. Test execution checkpointing for web applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 203–214.

[15] Salvatore Guarnieri and Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code.. In *USENIX Security Symposium*, Vol. 10. USENIX, Montreal, Canada, 78–85.

[16] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the World Wide Web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 177–187.

[17] Michael Hilton, Arpit Christi, Danny Dig, Michał Moskal, Sebastian Burckhardt, and Nikolai Tillmann. 2014. Refactoring local to cloud data types for mobile apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. ACM, 83–92.

[18] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: platform-independent memory debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 345–356.

[19] Young-Woo Kwon and Eli Tilevich. 2014. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering* 21, 3 (01 Sep 2014), 345–372. https://doi.org/10.1007/s10515-013-0136-9

[20] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 449–459.

[21] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 399–410.

[22] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[23] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning About JavaScript Promises. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 86 (Oct. 2017), 24 pages.

[24] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-driven Node.Js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 505–519. https://doi.org/10.1145/2814270.2814272

[25] Josip Maras, Jan Carlson, and Ivica Crnkovi. 2012. Extracting Client-side Web Application Code. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12)*. ACM, New York, NY, USA, 819–828. https://doi.org/10.1145/2187836.2187947

[26] med-chem rules. 2019. https://github.com/acarl005/med-chem-rules. (2019).

[27] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855711.1855722

[28] Marija Mikic-Rakic and Nenad Medvidovic. 2006. A classification of disconnected operation techniques. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*. IEEE, 144–151.

[29] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding Bugs by Isolating Unit Tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 496–499. https://doi.org/10.1145/2025113.2025202

[30] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 496–499.

[31] David A Patterson. 2004. Latency lags bandwith. *Commun. ACM* 47, 10 (2004), 71–75.

[32] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. 1996. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications* 4, 2 (1996), 63–71.

[33] Distributed Object Protocol. 2019. https://distributedobjectprotocol.org/. (2019).

[34] Realty rest. [n. d.]. https://github.com/ccoenraets/ionic2-realty-rest. ([n. d.]).

[35] recipebook. 2019. https://github.com/9bitStudios/recipebook. (2019).

[36] res postgresql. 2019. https://github.com/u4bi-sev/node-postgresql. (2019).

[37] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. 2016. Refactoring for software architecture smells. In *Proceedings of the 1st International Workshop on Software Refactoring*. ACM, 1–4.

[38] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 513–528.

[39] M. Selakovic and M. Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 61–72.

[40] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[41] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 842–853. https://doi.org/10.1145/2786805.2786830

[42] shopping cart. 2019. https://github.com/ComeAlongErica/full-stack-express-lab-shopping-cart. (2019).

[43] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/3106237.3106269

[44] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. 2016. Static DOM Event Dependency Analysis for Testing Web Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 447–459. https://doi.org/10.1145/2950290.2950292

[45] theBrownNode. 2019. https://github.com/clintcparker/theBrownNode. (2019).

[46] John Vilk and Emery D Berger. 2018. BLeak: automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 15–29.

[47] Xudong Wang, Xuanzhe Liu, Ying Zhang, and Gang Huang. 2012. Migration and execution of JavaScript applications between mobile devices and cloud. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 83–84.

[48] Huaming Wu, William Knottenbelt, Katinka Wolter, and Yi Sun. 2016. An Optimal Offloading Partitioning Algorithm in Mobile Cloud Computing. In *Quantitative Evaluation of Systems*, Gul Agha and Benny Van Houdt (Eds.). Springer International Publishing, Cham, 311–328.

[49] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. 2015. Uncovering JavaScript performance code smells relevant to type mutations. In *Asian Symposium on Programming Languages and Systems*. Springer, 335–355.