Client-Server Code Mobility at Runtime

Sebastian Heil*, Lucas Schröder and Martin Gaedke

Technische Universität Chemnitz, Chemnitz, Germany
E-mail: sebastian.heil@informatik.tu-chemnitz.de;
lucas.schroeder@informatik.tu-chemnitz.de;
martin.gaedke@informatik.tu-chemnitz.de
**Corresponding Author*

Received 15 September 2023; Accepted 21 March 2024

Abstract

Application logic is inherently distributed between client and server due to the fundamental Client/Server architecture of the Web. The individual distribution is specified at design time and remains unchanged stable, preventing individual load distribution between clients and server at runtime. Dynamic code mobility at runtime, in contrast, allows to balance the needs of users, through increased responsiveness, and software providers, through better resource usage and cost reductions. Enabled by WebAssembly, the Web ecosystem recently provides the technological foundation for relocating code units during runtime. However, leveraging these capabilities to enhance web applications with dynamic code migration presents challenges for web engineers. In response, we propose an innovative distributed Client/Server software architecture for web applications. This architecture facilitates the dynamic migration of code at runtime, and addresses the technical challenges like dependency management, control and data flow distribution, communication, and interfaces. This novel software architecture serves as a reference for web engineers aiming to enrich their web applications with dynamic code mobility. Additionally, it contributes to the ongoing reevaluation of

Journal of Web Engineering, Vol. 23_4, 507–534.
doi: 10.13052/jwe1540-9589.2342
© 2024 River Publishers

the Web ecosystem in light of the widespread adoption and standardization of WebAssembly across major browsers. Through experimentation in four scenarios, we demonstrate the feasibility of implementing this architecture, its negligible impact on performance and the optimization potential for individual code distributions across client and server.

Keywords: Web infrastructure, software architecture, code mobility, frontend, web user interface, WebAssembly, JavaScript, WebSockets.

1 Introduction

Contemporary Web applications are engineered based on a well-established stack of W3C-standardized technologies. The foundational Client/Server architecture of the Web inherently prompts software architects to consider the distribution of application logic between the two sides. The available design space is extensive, encompassing configurations that lean towards thin clients, with predominant execution of application logic on the server side, e.g. utilizing frameworks like Django, ExpressJS, Laravel, or Rails. At the other end of the spectrum, there are architectures in which a more substantial share of computations are ran on the client side in the browser, while the server offers only a simple interface to the underlying data layer, e.g. when developing using client-side frameworks such as React, Vue, Angular, or Svelte, coupled with extensive AJAX communication.

Determining an optimal allocation of application logic between client and server for a given web application is complex as it is influenced by variety of factors and project-specific requirements. What is more, the code distribution manually crafted by Web Engineers is static and fixed at design time. The assignment of code units to either the client or server is defined a priori and remains unmodifiable at runtime. This static design time code distribution curbs the capacity to respond to situational events and conditions dynamically. Particularly in view of the ever-increasing heterogeneity of user devices on the client side, this rigidity undermines the ability to support balancing responsiveness/usability requirements by users, with resource consumption and financial considerations on the part of software providers.

While deciding the right distribution for a given web application depends on various factors and individual requirements, the distribution is static and fixed at design time. The mapping of units of code to either the client or server side is decided a priori and cannot be changed later dynamically at runtime, allowing to react to situational events and conditions. Especially in light of

the ever-increasing heterogeneity of user devices on the client side, this static design time decision does not support balancing responsiveness/usability requirements by users, resource usage, and economic considerations by the software providers.

The emergence of server-side JavaScript through NodeJS, coupled with the ability to execute server-side languages on the client side via Web-Assembly [21], forges more homogenous client- and server-side platforms. This paves the way to achieve the vision of code mobility [1] in the Web at runtime. The resulting advantages include a dynamic and individual load distribution between clients and server, along with potential cost savings for software providers.

This article aims at devising a novel distributed Client/Server software architecture for web applications, which enables the dynamic reallocation of code execution at runtime. We address the technical challenges encompassing dependency management and compilation, control and data flow distribution, as well as the necessary communication and interfaces, proposing resolutions for each of them. The corresponding architecture, along with an infrastructure supporting Web Engineers to create web applications with dynamic code mobility, is both conceptualized and implemented, and subsequently subjected to testing across various experimental scenarios. This article extends our initial work on DCM presented at ICWE 2023 [10, 11], based on reviewers' comments and feedback throughout the conference, in particular in the following areas:

1. We enhanced the dependency management and simplified code distribution configuration for developers through automatic fragment id management and named fragments.
2. The DCM client-side infrastructure was extended to support Web Engineers to invoke mobile code units through automatic generation of awaitable JavaScript wrappers.
3. The DCM server-side infrastructure was extended to support Web Engineers to integrate DCM through customization of communication channel endpoints.
4. We extended experimentation for the fragment generation and compilation steps with now 5 codebases and the new fragment generation technique.
5. Additional experimentation to investigate the complete distribution space from all-client to all-server with regard to execution time impact was conducted.

6. We improved the experiment design to increase internal validity and replicated the previous experiments with the revised DCM infrastructure on updated hardware with significantly higher number of repetitions.

The remainder of this article is structured as follows: in section 2 we outline our proposed solution architecture and the supporting infrastructure, section 3 positions our work against existing code mobility paradigms and approaches, in section 4 we evaluate the feasibility of the architecture in 3 scenarios and show that the performance overhead is negligible, and section 5 concludes the article with an outlook on directions for future work.

2 The DCM Architecture

In this section, we introduce our solution aimed at facilitating dynamic code mobility between the client and server at runtime, utilizing W3C-standardized Web technologies. Our proposed software architecture, the DCM Architecture, empowers Web Engineers to integrate dynamic code mobility capabilities into their web applications, emphasizing minimal interference with established development activities. An overview of the primary components within the DCM Architecture and their interactions is depicted in Figure 1. The DCM architecture incorporates a dynamic migration infrastructure (highlighted in blue) that can be embedded within a web application (depicted in black), affording Web Engineers the means to manage code mobility through straightforward configuration. The DCM approach addresses three key challenges:

1. Specification and compilation of migratable code fragments for both the client and server side,
2. Orchestration of execution and control flow for these fragments between the client and server, and
3. Synchronization of fragment distribution information and redirection of data flow at runtime.

The following subsections detail our solutions to these three challenges.

2.1 Generation and Compilation of Code Fragments

Within this subsection, we draft our concept of mobile code fragments, explain how Web Engineers can define these segments of the codebase to be executable on both the client and server sides, discuss the necessary metadata and its semi-automatic extraction, as well as elaborate on the validation,

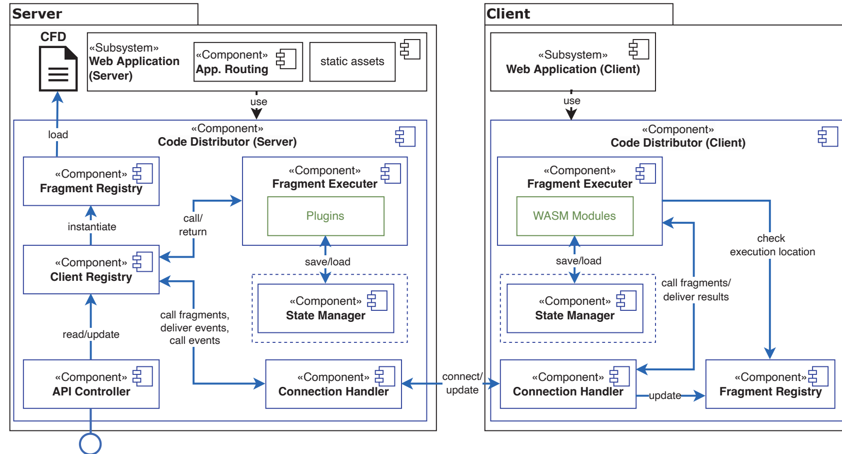

Figure 1 Main Components and Interactions of the DCM Architecture enabling Dynamic Code Migration between Client and Server at Runtime. Supporting Infrastructure is Highlighted in Blue, Automatically Generated Artifacts are Highlighted in Green.

compilation, and deployment of executable modules derived from specified code fragments.

2.1.1 Specification of Code Fragments

To enable Web Engineers to designate segments of a web application's codebase that can be dynamically relocated between the client and server during runtime, we introduce the concept of *Code Fragments*. A code fragment $CF = (D_i, L, T, M)$ encompasses the definition of its source document $D_i \in \mathfrak{C}$ within the codebase $\mathfrak{C}$, its limits $L = (\alpha, \omega)$ (line numbers $\alpha, \omega \in \mathbb{N}$) within D_i, its type $T \in \{function, variable, typedefinition\}$, and the migration-relevant metadata M. The limits can be expressed either through code annotations within the source code itself or via separate numerical specifications. In DCM, the level of granularity for specifying executable code fragments is at the function level, to achieve a balance between fine-grained control (smaller reuse units than components/classes) and isolation/dependency management (larger than sets of statements). Fragments for variables and type definitions handle imports of functions but are not independently executable.

Figure 2 illustrates the data model of a code fragment, combining information provided by Web Engineers and information automatically derived through static code analysis of the codebase. In addition to location and fragment type information D_i, L, T, Web Engineers define the intended initial

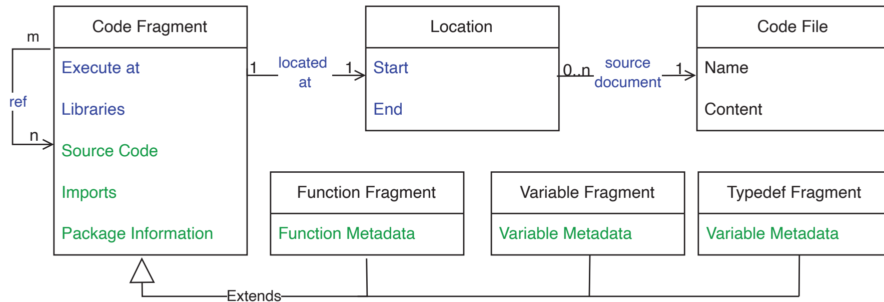

Figure 2 Code Fragments Data Model. Web Engineer provided information in blue, automatically identified information in green.

execution location (server or client), libraries employed, and referenced other fragments. The automatically identified information encompasses the actual source code as specified by the location information, imports from other sources, package information, and structural details of functions/variables/-type definitions such as function parameters. Fragment information for a web application implementing DCM is configured in a single additional artifact called code fragment description (CFD) in Figure 1. An excerpt of manually specified portions of such a CFD is presented in listing 1.

Through Syntax Analysis most of the information required for the compilation of code fragments can be automatically derived from the codebase. This significantly reduces the required effort for the Web Engineer by reducing the need for manual specification. For this, DCM analyses the *abstract syntax tree (AST)* which is created when parsing the code base, containing tokens and their relations according to the grammar of the programming language in use. Deriving the AST from a given source code is a language-dependent step. The Web Engineer's manual configurations in the CFD are enriched with data about imports from other sources, package information, and structure of functions/variables/type definitions gained through AST traversal.

In previous experiments, we observed some difficulties to manage the fragment IDs manually [10]. This is now addressed by using non-integer expressive fragment names as identifiers. The DCM infrastructure allows using named identifiers also for managing the depdency lists of a fragment.

```
1  fragments:
2  ...
3      - id: GetHash
```

```
 4        runOn: client
 5        location: name: { filepath: shared/shared.go }
 6        libs: [crypto/sha256, encoding/hex]
 7        ...
 8      - id: CreateEmployee
 9        runOn: client
10        dependsOn: [GetHash]
11        location: { filepath: shared/employee.go }
12    ...
```

Listing 1 Sample excerpt of a Code Fragment Description

2.1.2 Compilation of Code Fragments

To facilitate the execution of code fragments on the server and client side as defined by Web Engineers in the Code Fragment Description, these fragments must be compiled for the respective target platform. The server-side compilation target varies based on the specific language and platform, while the client-side target is WebAssembly. Compiled artifacts on the server side are treated as plugins, while those on the client side are treated as WebAssembly modules. Both binary artifacts can be loaded dynamically in their execution environments during runtime.

For building both the language-specific plugins and the WebAssembly modules, the source code, on which the compiler is then invoked, is automatically generated from the codebase and metadata of each code fragment. As the CFD partially comprises of user-provided information, a `CFD Validator` assesses the structural integrity and completeness of the CFD. Errors such as duplicate fragments or missing essential information are reported, accompanied by debug information to assist Web Engineers in resolving them. Upon successful validation, automated code generation and transformation take place. The generated code handles imports and dependencies, rendering them as units of code that can be compiled independently. This process considers metadata for all fragments as duplicate imports resulting from dependency chains necessitate resolution. The existing codebase is modified to reroute invocations of migratable fragments to the `Fragment Executer`. This permits execution of either server-side plugins or forwarding of control and data flow to the client-side DCM infrastructure, where it is executed by corresponding the WebAssembly module. To improve developer experience over the initial /approach version [10], the infrastructure now automatically generates JavaScript wrappers for all callable fragments

specified in the CFD. As shown in Listing 2 line 2, the Web Engineer can import these wrappers in the client-side application code even though they do not exist at that moment. The wrappers are implemented as awaitable JavaScript functions. The compilation automation then injects components of the DCM infrastructure (e.g., `Code Distributor` in Figure 1). The adjusted codebase is subsequently compiled for both server and client, and plugins and WebAssembly modules (including JS glue code) are placed in the appropriate directories to be available for server and client-side code distributor instances. Specifically, WebAssembly modules must be deployed within the statically served assets directory, alongside JavaScript and CSS files, for retrieval via HTTP(S) on the client side.

```
1  // Import of generated JS Wrappers
2  import {ValidateCouponInput, ValidateCoupon}
       ↪ from './CodeDistributor/functions.js';
3
4  //Invocation of JS Wrapper ValidateCoupon
5  #validateCoupon = async () => {
6    try {
7        const valid = await ValidateCoupon(this.
           ↪ input.value)
8    } catch (err) {
9      // error handling omitted
10   }
11   // results processing omitted
12 }
```

Listing 2 Client-side invocation of a code fragment

2.2 Dynamic Migration of Code Fragment Execution

To enable dynamic changes in the execution location of code fragments during runtime, the DCM architecture incorporates infrastructure to oversee their life cycle, as well as the distribution of control and data flow. These responsibilities are implemented by the `Code Distributor` components on both the server and client side (see Figure 1). These components manage loading, execution, and termination of plugins/WebAssembly modules, while ensuring the transfer of incoming and outgoing data flows and events.

2.2.1 Code Distributor (Server)

The `Code Distributor` component can be either embedded within the web application itself or exist as an external, stand-alone server process, potentially on a different host. In contrast to approaches like HTML5 Agents [20], the DCM architecture advocates direct embedding due to lower resource requirements, and operational/maintenance demands, as well as reduced communication complexity. Embedding entails adding the DCM library to the web application's imports and configuring routes for the `Code Distributor` within the application's internal routing. These steps can be automated through the codebase adjustments outlined in Section 2.1. The `Code Distributor` validates incoming client requests and signals connection errors. For valid requests, it manages sessions through the `Client Registry`. `Client Registry` and `Fragment Registry` together monitor all code fragments and enable individual fragment distribution patterns for each client. Both registries are populated based on the Web Engineer configuring the CFD to specify available code fragments and their initial execution locations. The distribution status of all fragments is synchronized with each client's `Code Distributor`, allowing updates in fragment execution location to trigger control and data flow migration through the `Fragment Executer`. For server-side execution, the `Fragment Executer` loads and executes the corresponding compiled plugin fragment. Fragment state is handled by the `State Manager` to enable restoration after migration. This encompasses modified/initialized variables, loops, and time functions, along with synchronized resource changes shared among other fragments. A RESTful interface provided by the `API Controller` allows monitoring and controlling the dynamic code mobility. The endpoint URL can be customized as detailed in Section 2.3 for the WebSocket connection. The interface outputs the invidual fragment distribution of a client and allows adjustment of this distribution at runtime. Potentially, this API supports connecting automated decision-making based on runtime metrics like load and network bandwidth to optimize fragment distributions.

2.2.2 Code Distributor (Client)

The `Fragment Executer` and `State Manager` components handling execution and state management as detailed above are mirrored on the client-side. In contrast to the server-side `Code Distributor` that has the same programming language as the web application's backend, its counterpart on the client side is implemented in JavaScript and needs to be statically served through the application's web assets. The `Fragment Executer`

manages loading, initialization, and invocation of fragments compiled as WebAssembly modules, along with data conversions between JavaScript and the backend language'S type system within the WebAssembly modules. Execution information is read from the `Fragment Registry` prior to each invocation, which synchronizes with the server-side counterpart. Similar to offloading approaches such as MAUI [4] and ThinkAir [13], fragments are initially executed locally. This state remains until updates are received, enabling execution during connection establishment and initialization. Similarly, when connection between to the server side is lost, the DCM infrastructure falls back to local execution at the client side. To execute a fragment, a pool of WebWorkers[1] is used.

2.3 Client/Server Synchronization and Data Flow Redirection

To maintain the bidirectional exchange of information necessary for synchronizing fragments and distribution state, as well as to facilitate data flow from and to fragments when their execution location changes, the DCM code distributor components require a constant means communication. This subsection outlines the communication channels and protocol. We employ WebSockets for communication, which offers bidirectional connectivity between the client and server-side DCM infrastructure, and is more widely supported by browsers than the new WebTransport W3C standard. The updated DCM infrastructure now allows Web Engineers to customize the endpoints used for the WebSocket channel. This improves integratability with the existing code base of the web application, enabling to select the most suitable URL for the application-internal routing of messages to the DCM infrastructure. Along with the endpoint for the `API Controller`, the URL can be specified in the DCM configuration and the infrastructure will perform the necessary changes to register the routes with the web application during the automatic code modifications described above. JSON is used as message format within the WebSocket connections. The `Connection Handler` components connect with each other via WebSockets during initialization. Clients identify themselves via JSON Web Tokens (JWT) included in all communications to enable the server to manage fragment distribution for each client individually. Client-server communication is implemented via JSON-encoded events exchanged over the WebSocket connection.

[1] cf. https://html.spec.whatwg.org/multipage/workers.html

Table 1 DCM Communication Protocol

Name	Payload	Description
updateFragments	[object] fragmentStatusList	triggered whenever there is a change in the fragments' distribution to synchronize Client and Server Distributor
callFunction	string funcName, [object] params, boolean defer?	triggered when a fragment is called remotely, to pass incoming data to the called fragment
functionResult	string funcName, object result	triggered when a fragment invocation yields a result to return it to the caller

Table 1 presents the primary events within the DCM communication protocol, serving two key purposes: a) exchanging information necessary for fragment management and distribution state, and b) enabling data flow redirection for fragments where the caller and callee are not on the same side.

The codebase modifications delineated in section 2.1 allow both client and server `Fragment Executer` components to act as intermediaries between the caller of a fragment and the fragment code itself, similar to the proxy pattern in traditional RPC. Data flows in and out of fragments via function parameters, return values, and global variables. The CFD represents information about these aspects, as described earlier. During runtime, each `Fragment Executer` verifies the current execution location of the called fragment and, if the location is remote, redirects the data flow. The `callFunction` event includes an optional `defer` property, signaling to the `Fragment Executer` to execute the invoked fragment in the background. This accommodates long-running computations in the originally called fragment, while permitting other fragments to execute concurrently.

3 Related Work

The concept of code mobility has long been a subject of research interest in distributed systems [1]. While the predominant code mobility paradigm on the Web is *Code on Demand*, the availability of NodeJS and the WebAssembly standard [21] have opened paths to other paradigms like *Remote Evaluation* and *Mobile Agents* in the current Web environment. Table 2 provides an overview of the current code mobility landscape.

Code on Demand [1] stands as the most widely used code mobility paradigm in Web applications, allowing clients to request and execute code from the server at runtime through HTML script tags that implement

the loading and execution of JavaScript files within the browser. Popular client-side frameworks like React, Vue, Angular, or Svelte are built upon this architecture, where JavaScript code is loaded dynamically, often aided by Content Delivery Networks (CDNs) for efficient delivery of commonly used code artifacts for these frameworks. Notably, the paradigm Code on Demand is included as the sixth architectural constraint of REST [6], thus influencing the architectural design of many contemporary Web applications. Sparkle [19] goes one step beyond common Code on Demand practice by supporting also to capture, migrate, and restore application state. In contrast to the DCM architecture, Code on Demand mobility is unidirectional, as it exclusively migrates code units from the server to the client. While the name "on demand" implies a dynamic level of mobility, this is only partially fulfilled: code artifacts are re-distributed from the server to the client at runtime, but the actual decision to do so is established at design time and remains fixed in typical Web applications using JavaScript or more recent platform-specific approaches like Blazor[2].

Remote Evaluation [1] mobility facilitate the one-way mobility of code from the client to the server. Focused on supporting resource-constrained mobile devices, offloading approaches such as MAUI [4], CloneCloud [3], or ThinkAir [13] shift complex computations to the server. Code offloading has become particularly relevant in the context of cloud computing [17]. Similar to DCM, MOJA [2] and PIOS [18] leverage a uniform platform across the client and server side, with MOJA also using WebSockets for communication. However, unlike the DCM architecture, these approaches exclusively enable code migration from the client to the server. Additionally, due to their reliance on NodeJS, they are limited to JavaScript, while DCM's use of WebAssembly modules allows for the potential use and migration of code written in various other Web languages.

Both Code on Demand and Remote Evaluation approaches do not consider data flow redirection or state transfer, as their primary focus is on unidirectional code mobility. This is addressed by the third following code mobility paradigm.

Mobile Agents [1] approaches aim at the mobility of entire software components across the network at runtime. An early example is Telescript [5], which enabled mobile agents using a dedicated object-oriented language supporting the migration of objects as software agents to other *places* at runtime. Telescript also includes the capability to interrupt the execution

[2]https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

of an agent and resume it in the new place. Without the requirement for using a dedicated new language, many mobile agent approaches, such as Java Aglets [15], make use of Java's threading and networking capabilities. Aglets combine Java Aplets and Servlets, specifying lifecycle methods for Java objects to support creation, cloning, dispatching, retraction, activation/deactivation, and messaging, allowing them to move between the client and server. The execution of Aglets can be paused and resumed with the previous state restored in the new location.

As a more standards-based approach, HTML5 Agents [20] implements the mobile agents paradigm using standardized Web technologies like HTML5, CSS, and JavaScript, capitalizing on platform uniformity between client and server through NodeJS. Additionally, newer Web standards such as WebRTC and WebWorkers are leveraged in contemporary mobile agents frameworks such as Liquid.js [9]. It focuses on delivering a seamless user experience for moving Polymer.js-based Web Components across multiple heterogeneous devices, effectively allowing these components to "follow" the user. Challenges in cross-device liquid computing specific to the JavaScript runtime , especially in handling closures, have been addressed by Disclosure [12]. To facilitate the execution state of migrated components, it proposes an instrumentation-based technique with a limited runtime performance impact of 0-15%. Mobile agents approaches are employed for code mobility in the cloud-edge continuum. A recent approach is Self-distributing Systems (SDS) [8], which puts particular emphasis on the state management of re-locatable components [7]. The security challenges intrinsic to mobile agents require additional cryptographic techniques to be implemented as in [14].

Compared to DCM, mobile agent approaches operate at a lower level of granularity, moving entire components, including their code. Once a component is moved, it no longer exists at its original location and can freely move between peer hosts. This behavior potentially implies design-level security challenges associated with mobile agents [1]. In DCM, these challanges are less prominent, since code mobility occurs only within the client/server boundaries of the same Web application, and the duplicated existence of the binary code fragments on client and server side make it much harder to introduce malicious "new code." The limited success of the mobile agents paradigm has also been attributed in part to the lack of an established platform, particularly within browsers [1]. DCM leverages the changed landscape with widespread browser support for WebAssembly, akin to the role of containers in [16].

Table 2 Code Mobility landscape

Decision \ Mobility	Unidirectional	Bidirectional
Design Time	Code on Demand HTML script tags, REST, .NET Blazor, Sparkle [19]	N/A
Runtime	Remote Evaluation MAUI [4], CloneCloud [3], ThinkAir [13], MOJA [2], PIOS [18]	Mobile Agents Telescript [5], Aglets [15], HTML5 Agents [20], Liquid.js [9], Disclosure [12], SDS [8]

4 Evaluation

To assess the DCM architecture and accompanying infrastructure, we implemented and instantiated them using the Go language platform and its compiler toolchain. Our experimentation encompasses four distinct scenarios, each designed to test different facets of the architecture and infrastructure. These experiments aim to scrutinize the performance implications of our proposed solution. Below, we outline the experimental materials and methodology, present the results, and discuss the gained insights.

4.1 Material and Procedure

Our evaluation implementation of DCM, based on the Go language platform, is available online for review. To ensure consistent conditions for repeated evaluation runs and measurements, we containerized the evaluation setup using Docker.

Our experimental evaluation encompasses four scenarios, each with different configurations. These scenarios collectively examine the performance impact of our proposed solution. Scenario I involves the steps for analysis and compilation of code fragments as executable modules described in section 2.1. Scenario II focuses on the network communication implementing the DCM protocol between the client and server side Code Distributor components, assessing stability and delays. Scenario III is an evaluation of the runtime behavior of migration and fragment execution. Scenario IV explores

Table 3 Experimental hardware environment

Side	CPU	RAM	OS
Server	Intel Core i5-4460 4 Cores @3.2GHz	16GB DDR3-1600	Debian Linux 12.1 64 bit
Client	AMD Ryzen 3 5425U 4 Cores @2.7GHz	24GB DDR4-3200	Debian Linux 11.7 64 bit

Table 4 Materials used for scenario I, indicating number of Go source files (#d) and number of code fragments (#cf)

Name	#D	#CF	Source
fzf	59	19	https://github.com/junegunn/fzf
Gin	92	25	https://github.com/gin-gonic/gin
frp	200	20	https://github.com/fatedier/frp
terraform	1240	7	https://github.com/hashicorp/terraform
custom	7	38	https://github.com/heseba/dcm

the impact of different fragment distributions. All evaluation materials and test scripts are provided online[3].

The test runs of the scenarios were using the hardware setup shown in Table 3. Network connection between client and server via Wi-Fi 4 (IEEE 802.11n) with a delay between server and client measured via ping ranging between 1.2ms and 40.0ms around a mean of 32.1ms ($\sigma = 8.5$ms).

4.1.1 Scenario I

This scenario explores the code fragment analysis and compilation steps, assessing how different codebase sizes affect each component involved. As material for scenario I, we selected four popular open-source Go projects having a minimum ranking of 1000 stars on GitHub of different complexities for evaluation as shown in Table 4. In addition to the four public projects, we developed a custom test suite in Go, consisting fewer Go sources files but with a high number of fragments designed explicitly to test various computations, errors, and data type handling. For all five sample projects, we manually crafted the code fragment descriptions. To simulate integration into a web engineer's project settings, we created Docker configurations. The test script executes three automated steps: *code analysis*, *fragment generation*, and *compilation*. Each step underwent five repetitions, and execution times were recorded.

[3] https://github.com/heseba/dcm

4.1.2 Scenario II

In this evaluation scenario, network connections and communication timings between the server and client sides of the DCM architecture are analyzed. Three different timings are captured: the round-trip time for an echo signal between the client and server, the time taken to establish the initial WebSocket connection on the client and for it to receive the first fragment list update, and the time interval between receiving a migration command via the server-side interface and the client receiving the new distribution information. To facilitate these measurements, we developed a custom test application using a modified version of the DCM infrastructure with extended access to the WebSocket connection and built-in time measurements. Specifically, we measure the following three timings:

1. *Echo time* is measured by having the client send an echo event to the server until receiving a response.
2. *Initialization time* measurement is initiated by reloading the page on the client, which requests the list of fragments and execution locations until it is received.
3. *Command time* measures the duration from the server-side reception of a location update command via the API, forwarding this information to the client until it is successfully received.

All three time measurements are repeated 100 times, resulting in 300 individual test runs.

4.1.3 Scenario III

This scenario serves as a demonstration of migration and fragment execution at runtime. It tests the execution results both before and after executing fragments on the server and client side. To emulate long-running, side-effect-free computations, we implemented two algorithms in Go and JavaScript: Fibonacci and nth prime. Both algorithms were executed in three versions: as server-side fragment plugins, as client-side WebAssembly modules, and in plain JavaScript as the baseline for comparison. Scenario III encompasses three distinct test cases:

1. *Single* evaluates the computation of Fibonacci to $n = 100$
2. *Iterated* assesses the computation of Fibonacci to $n = 93$ repeated in a loop 1000 times
3. *Prime* examines the behavior under optimized and non-optimized execution conditions by calculating the 500,000th prime number

The value of $n = 93$ for the "Iterated" test case is defined by the limit of Go's `int64` type. The optimization of computation conditions for the "Prime" test case is implemented as follows: under optimized conditions 50 iterations are run in a loop within the fragment itself, under the non-optimized conditions, the fragment is invoked 50 times from outside. During the longer executions, the system's behavior when receiving incoming migration commands was monitored.

4.1.4 Scenario IV

This scenario investigates the performance impact of differing fragment distributions for one client-server pair in order to explore the optimization potential of location changes of individual code fragments. To that end, scenario IV fully covers the distribution space from the all-client to all-server configurations. As material, we use a custom codebase which contains $n = 5$ code fragments, which form an execution sequence: the result of fragment CF_1 invocation is passed to CF_2, the result of which is passed to CF_3 and so on. As each fragment can have one of two possible execution locations $E(CF_i) \in \{server, client\}$, the scenario comprises $2^n = 2^5 = 32$ different configurations for which corresponding CFDs are created. Each configuration is executed with 100 repetitions, resulting in $\#r = 3200$ test runs to measure the execution times for all possible fragment distributions.

4.2 Results

In this section we report the results of our experimentation in the four scenarios. Please note that even for scenarios I-III these results differ from our previous experimentation in [10], as all experiments were re-run with the extended DCM infrastructure, on a new hardware setup as in Table 3, and with higher numbers of repetitions.

Scenario I. Table 5 shows the statistics of the time measurements for scenario I for the five samples per each step collected in 5x100 test runs. A docker container was used to execute the test runner script. CFD specification was facilitated by the DCM infrastructure enabling named fragments and linting functionalities for missing dependencies or attributes. The new dependency management technique did not interfere with the successful completion of all 500 test runs: all corresponding fragments were compiled and deployed automatically.

Scenario II. The measured times for the network communications of scenario II for each test case collected in 100 test runs are shown in Table 6.

Table 5 Scenario I Time measurements: min t_{min} and max t_{max}, mean μ, median $\tilde{t}$ and std. deviation σ

Sample	Step	t_{min}	t_{max}	μ	$\tilde{t}$	σ
fzf	code analysis	116ms	231ms	124.7ms	122ms	12.0ms
	frag. generation	128ms	157ms	131.3ms	131ms	3.0ms
	compilation	8,751ms	9,393ms	8,835.6ms	8,818ms	81.0ms
Gin	code analysis	127ms	271ms	136.1ms	134ms	14.5ms
	frag. generation	127ms	224ms	131.8ms	131ms	9.9ms
	compilation	5,769ms	28,337ms	6,058.0ms	5,826ms	2,250.7ms
frp	code analysis	194ms	267ms	210.9ms	208ms	10.971ms
	frag. generation	129ms	160ms	132.8ms	132.5ms	3.284ms
	compilation	7,033ms	7,405ms	7,113.9ms	7,087ms	80.1ms
terraform	code analysis	2,184ms	2,704ms	2,247.6ms	2,242ms	51.414ms
	frag. generation	298ms	403ms	303.2ms	302ms	10.3ms
	compilation	698ms	808ms	712.0ms	710ms	13.6ms
custom	code analysis	6ms	23ms	6.2ms	6ms	1.7ms
	frag. generation	123ms	152ms	126.1ms	125ms	3.1ms
	compilation	12,109ms	12,825ms	12,195.3ms	12,162ms	135.3ms

Additionally, connection recovery behavior was tested. When the page was reloaded on the client side, the system could successfully reconnect the server and client `Code Distributor` components in presence of several other connected clients with a mean time to recover from a connection loss of 15s.

Table 6 Scenario II Time measurements: min t_{min} and max t_{max}, mean μ, median $\tilde{t}$ and std. deviation σ

Time Measurement	t_{min}	t_{max}	μ	$\tilde{t}$	σ
Echo time	2 ms	48 ms	32.14 ms	34 ms	7.844 ms
Initialization time	28 ms	61 ms	34.95 ms	34 ms	5.591 ms
Command time	23 ms	42 ms	31.36 ms	31 ms	3.700 ms

Scenario III. Table 7 shows the time measurements of the three test cases in different configurations. All measurements are with browser-side caching enabled and excluding the initial loading times of WebAssembly modules. These were measured at a mean of 44ms ($\sigma = 12ms$). Fragments could be successfully migrated between server and client. Computed fragment execution results of WebAssembly modules and server plugins were always identical.

Scenario IV. Figure 3 shows the time measurements of the 32 test cases across 100 test runs. The detailed data in Table 8 shows the descriptive statistics for the different client-server fragment distributions, ordered from

Table 7 Scenario III Measurements: number of elements and iterations and overall execution times for JavaScript, WebAssembly and Server plugins

Case	Elem.	Iter.	JS	WASM	Plugin
Single	100	1	1 ms	113 ms	37 ms
Iterated	93	10	1ms	1,083ms	107ms
	93	1,000	12ms	101,049ms	2,766ms
Prime	100,000	10	0.827s	1.454s	2.815s
	500,000	50	43.804s	41.591s/ 33.136s*	104.389s

* optimized

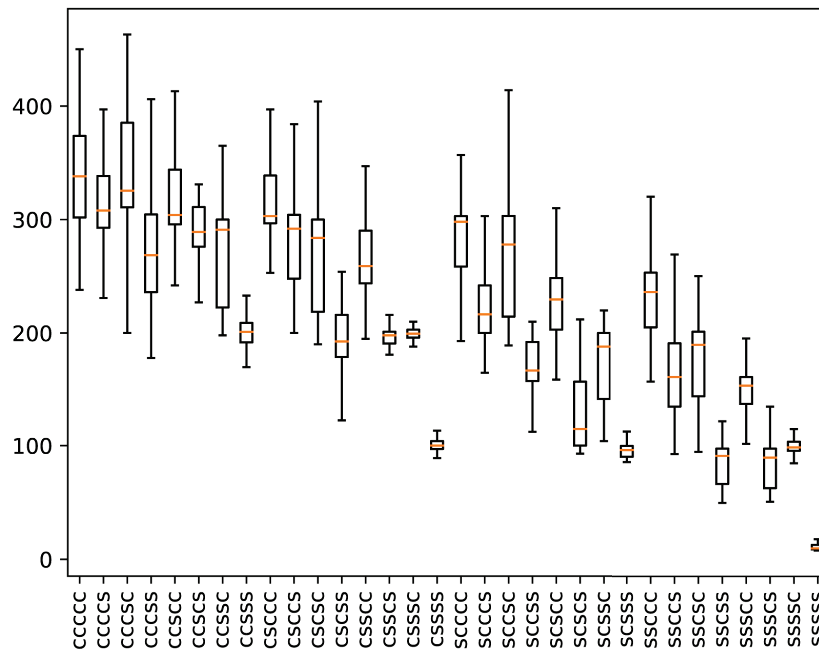

Figure 3 Scenario IV execution times of 32 fragment distributions outliers filtered.

all-client to all-server. The distribution column indicates the execution location of the five code fragments: $dist = E_1E_2E_3E_4E_5$ with $E_i \in \{c, s\}$ such that $E_i = c \iff CF_i$ is executed on the client and $E_i = s \iff CF_i$ is executed on the server. The five fragments are in the order of their execution sequence, i.e. CF_1 is the first to be executed and CF_5 the last with their computation results passed in sequence.

Table 8 Scenario IV Time measurements: code fragment distribution $dist$, min t_{min} and max t_{max}, mean μ, median $\tilde{t}$ and std. deviation σ

$dist$	t_{min}	t_{max}	μ	$\tilde{t}$	σ
ccccc	238 ms	896 ms	350.66 ms	338 ms	90.166 ms
ccccs	231 ms	489 ms	314.42 ms	308 ms	41.125 ms
cccsc	197 ms	530 ms	343.47 ms	325.5 ms	59.524 ms
cccss	178 ms	905 ms	294.74 ms	268.5 ms	121.114 ms
ccscc	242 ms	526 ms	325.12 ms	304 ms	52.097 ms
ccscs	197 ms	538 ms	288.04 ms	289 ms	40.834 ms
ccssc	198 ms	365 ms	269.54 ms	291 ms	41.099 ms
ccsss	114 ms	742 ms	203.59 ms	201 ms	62.221 ms
csccc	253 ms	556 ms	323.49 ms	303 ms	59.490 ms
csccs	200 ms	1181 ms	305.61 ms	292 ms	137.912 ms
cscsc	190 ms	452 ms	270.74 ms	284 ms	52.546 ms
cscss	112 ms	420 ms	198.25 ms	192.5 ms	41.293 ms
csscc	195 ms	362 ms	263.39 ms	259 ms	34.050 ms
csscs	107 ms	216 ms	187.33 ms	198 ms	28.801 ms
csssc	104 ms	1009 ms	206.39 ms	199.5 ms	87.475 ms
cssss	78 ms	127 ms	100.3 ms	100 ms	8.631 ms
scccc	193 ms	388 ms	283.46 ms	298 ms	38.197 ms
scccs	165 ms	311 ms	219.79 ms	216.5 ms	26.772 ms
sccsc	189 ms	627 ms	289.45 ms	278 ms	104.367 ms
sccss	98 ms	210 ms	168.65 ms	167 ms	27.803 ms
scscc	159 ms	321 ms	232.21 ms	229.5 ms	35.552 ms
scscs	93 ms	212 ms	131.56 ms	115.5 ms	35.306 ms
scssc	104 ms	220 ms	170.52 ms	188 ms	34.666 ms
scsss	51 ms	139 ms	90.64 ms	96.5 ms	18.421 ms
ssccc	157 ms	635 ms	246.46 ms	236 ms	72.509 ms
ssccs	93 ms	403 ms	163.55 ms	161 ms	43.592 ms
sscsc	95 ms	250 ms	172.9 ms	189.5 ms	35.040 ms
sscss	50 ms	122 ms	83.76 ms	91.5 ms	18.151 ms
ssscc	91 ms	209 ms	149.04 ms	153.5 ms	31.032 ms
ssscs	51 ms	379 ms	89.26 ms	90 ms	45.037 ms
ssssc	57 ms	135 ms	98.28 ms	99 ms	11.285 ms
sssss	8 ms	48 ms	11.91 ms	10 ms	5.740 ms

4.3 Discussion

Scenario I. The measured times show that the analysis and compilation steps depend on the codebase complexity. Intuitively, the more source files there are in the codebase, the longer the accumulated time for AST creation and thus the longer the code analysis time as well as the longer the time to generate the fragment code, as the required number of comparisons between the CFD and the fragment code increases. The correlation between the number of

source files in the codebase $\#D$ and the measured analysis times (Spearman's $\rho = 0.97$, $p < 0.001$) and generation times $\rho = 0.84$, $p < 0.001$) are highly significant at $\alpha = 0.01$. Even for the largest codebase in our experiments, Terraform, which comprises about 1200 Go source files, the analysis times remain well below 3 seconds and the time for fragment generation well lower than 1 second. The third step measure, the compilation of the resulting modified codebase, directly depends on the number of code fragments produced in the generation step. The correlation between the number fragments $\#D$ and the measured compilation time (Spearman's $\rho = 0.59$, $p < 0.001$) are highly significant at $\alpha = 0.01$. Compiling the custom codebase comprising of 38 fragments took about 17 times longer than the 7 fragments of terraform. Overall, we consider the impact on normal compilation and deployment activities for Web applications through the DCM infrastructure negligible. In particular, these extra times are not a runtime penalty, but occur at design time and therefore only when the codebase is modified and without negative impact on the user experience. During experimentation with the 5 test cases of scenario I, the DCM infrastructure's linting functionality automatically proposing corrections facilitated the execution of the experiment for the researcher previously unfamiliar with DCM.

Scenario II. During our experiments evaluating the network communication and stability of DCM, the WebSocket connections between the client- and serverside `Code Distributor` instances were established and managed reliably for several clients. When network errors occured, the automatic local execution and re-connects took over in order to ensure a consistent behavior of the user interface. The mean times measured for all three test cases of about 33 ms in 300 test runs are approximately equal to the raw network delay of 32 ms of the test setup and with a comparable standard deviation. Therefore, the performance impact of implementing the DCM architecture in a Web application through additional network communication is very low and not noticeable by end users.

Scenario III. The execution times for WebAssembly modules on the client side are higher in comparison to native JavaScript and the server plugins for two of three test cases. Here, the impact of the additional loading time of WebAssembly modules, even with caching enabled, can be seen. However, when considering that the mean loading time is 48 ms, the execution times for the WebAssembly modules are much closer to those measured for the server plugins. JavaScript consistently outperforms WebAssembly and plugins for all but the largest test case. This advantage especially for lower numbers of elements and iterations roots in the speculative optimizations of V8 runtime's

JIT compiler Turbofan, that creates shapes for monomorphic functions. As the number of elements and iterations increases, the advantage of JavaScript over the two fragment types decreases, from a ratio of 1:113 for the Single test case to 1:0.95/0.76 for the largest Prime test case. In these situations, the effect of executing a compiled language instead of interpretation leads to lower times for the server-side plugins and to comparable times for the WebAssembly modules.

Scenario IV. The results experimentation with 32 different code fragment distributions show that there is a clear difference between the execution times from a minimum of 12.0 ms for the all-server to a maximum of 350.7 ms for the all-client distribution. The Kruskal-Willis test shows that this difference is highly significant at $\alpha = 0.01$ ($H(31) = 2,698, p < 0.001$). This means that the specific selection of execution locations for a given set of fragments has an impact on the system performance, indicating a good potential for optimization. In line with scenario III, our data shows that execution on the server was faster than on the client. This can be seen in the significant ($\alpha = 0.01$) strong negative correlation between the execution times and the number of fragments executed on the server side (Spearman's $\rho = -0.892$, $p < 0.01$). All measurements were performed on the test setup in idle state, so further analysis simulating different load situations can investigate the concrete optimization potential depending on the situationally available client and server resources.

4.4 Threats to Validity

The experiments presented above are designed to offer a proof of concept for the feasibility of the DCM architecture and provide initial insights into its implementation based on the DCM infrastructure.

Internal validity could potentially be compromised by the researcher executing them. Compared to our initial experimentation in [10], we reduced this bias through execution by a researcher who was previously not familiar with the architecture and toolchain. A potential bias may stem from a single person executing the complex experimentation, but it should be noted that our claims do not pertain to the experience, effort, or difficulty faced by Web Engineers. Supporting such claims would necessitate a user study involving developer test subjects with diverse demographics. Moreover, the specific selection of public real-world projects as material in scenario I might have influenced the results. We addressed this by choosing four popular Go projects from GitHub, significantly varying in their size and also stemming from different application domains. The time measurement procedure

eliminates the risk for subjective biases, as all measurements were collected through automated means integrated into the scenarios' code. Additionally, all materials required for replication of this study are available on GitHub.

External validity of our experiments is constrained by the choice of the Go ecosystem. While this choice is valid for demonstrating the feasibility, the concrete measurement results cannot be extrapolated to other WebAssembly-compilable languages. This limitation is due to code analysis, fragment generation, and compilation being reliant on the available AST parsers, compilers, and language features. Consequently, further experimentation with other Web languages is necessary to gain a broader understanding of the DCM architecture. Generalizing our results beyond feasibility, particularly regarding the applicability of DCM in different application domains, is not within the scope of this study and would necessitate dedicated experiments employing qualitative empirical approaches.

Construct validity is limited with regard to the measurements of command time in scenario II. Unlike the measurements in scenarios I, III and IV, as well as the echo time and initialization time measurements in scenario III, the command time calculation uses the difference between start and end time stamps from two different host systems. While a more thorough examination of performance would demand sophisticated instrumentation to synchronize system clocks of the client and server host system, we assert that the command time measurements are still valid for demonstrating the general magnitude of DCM's impact. Also, we refrain from making specific numerical claims for these measurements beyond. Both hosts' clocks were synchronized via the Network Time Protocol (NTP). According to the NTP documentation[4], the expected precision lies in the range of 5-100ms. Thus, even in the presence of significant synchronization differences between server and client during our measurements, command times would not exceed 250ms, which is still in line with our claims of low performance impact as perceived by end users.

5 Conclusion & Future Work

This article presented a novel software architecture for Web applications that facilitates the dynamic mobility of code fragments during runtime. This architecture allows for distinct fragment distributions tailored individually to each client-server pair, enabling better adaptation to the situational availability

[4]cf. http://www.ntp.org/ntpfaq/NTP-s-algo.htm

of client and server resources. Unlike earlier code mobility approaches, the architecture is not bound to a specific platform such as JavaScript or Java and instead leverages W3C-standardized and well-established technologies such as WebAssembly and Web Sockets.

On the basis of extensive experimentation with the proposed DCM architecture, we shared our findings, addressing the principal challenges of specifying and compiling mobile code fragments, orchestrating control flow for fragment execution, and managing fragment distribution while redirecting data flow at runtime. To support Web Engineers in building Web applications with dynamic code mobility based on DCM, we devised a complementary infrastructure. The proposed architecture and infrastructure are evaluated through experimental results from three distinct scenarios that cover various architectural aspects and assess the performance impact of the solution. We have made all implementation and experimental materials available to empower the Web Engineering community to replicate our experiments and extend the approach.

The experiments produced a proof of concept for the feasibility of creating Web applications with dynamic code mobility following the DCM architecture and offered initial insights into its implementation with the support of the proposed infrastructure. Furthermore, they indicate that the performance impact resulting from the necessary fragment management and communication can be limited to negligible levels.

While these results are promising and highlight the potential of the WebAssembly standard for enhancing the capabilities of the current Web application infrastructure in terms of code mobility, we have identified several limitations and areas for future research. Notably, as functions become more stateful, handling state management during runtime migration becomes increasingly challenging, which is intrinsic to code mobility [1]. The design of appropriate mechanisms for interrupting and resuming long-running functions while preserving internal states presents an ongoing challenge, the feasibility of which depends on specific language platforms.

Additionally, due to current limitations in exchanging data between JavaScript and WebAssembly modules as numerical data via the Heap, handling fragments with data flows comprising complex structured data that cannot be easily represented necessitates the development of more advanced serialization techniques or potential future extensions of the WebAssembly standard to support object transformations.

In a broader conceptual sense, our work establishes a foundation for dynamic mobility of code fragments at runtime. To fully harness the benefits

of dynamically balancing user responsiveness and usability requirements with resource utilization and economic considerations by software providers for individual client devices, an automatic decision system is essential. Such a system could optimize fragment distribution at runtime based on information about individual hardware capabilities and, especially, by runtime load measurements on the client and server sides, and interact with DCM via the provided API. The creation of such a decision system and its integration with dynamic code mobility as in DCM represents a promising direction for future research, and we intend to contribute presenting initial insights from ongoing experiments in this direction.

Acknowledgements

The authors would like to thank Alexander Senger for his valuable contributions to the initial version of the proof-of-concept implementation of the DCM architecture and the evaluation experiments.

References

[1] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Is Code Still Moving Around? Looking Back at a Decade of Code Mobility. In *Proc. of ICSE'07 Companion*, pages 9–20. IEEE, 2007.

[2] Chaoran Xu, Niall Murray, Yuansong Qiao, and Brian Lee. MOJA - Mobile Offloading for JavaScript Applications. In *Proc. of ISSC/CIICT 2014*, pages 59–63. Institution of Engineering and Technology, 2014.

[3] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proc. of 6th EuroSys*, page 301, New York, New York, USA, 2011. ACM Press.

[4] Eduardo Cuervo, Aruna Balasubramanian, Dae-Ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. of 8th MobiSys*, page 49, New York, New York, USA, 2010. ACM Press.

[5] P. Domel. Mobile Telescript agents and the web. In *COMPCON '96. Technologies for the Information Superhighway Digest of Papers*, pages 52–57. IEEE Comput. Soc. Press, 1996.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[7] Roberto Rodrigues Filho, Luiz F. Bittencourt, Barry Porter, and Fábio M. Costa. Exploiting the potential of the edge-cloud continuum with self-distributing systems. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 255–260, 2022.
[8] Roberto Rodrigues Filho, Renato S. Dias, João Seródio, Barry Porter, Fábio M. Costa, Edson Borin, and Luiz F. Bittencourt. A self-distributing system framework for the computing continuum. In *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10, 2023.
[9] Andrea Gallidabino and Cesare Pautasso. The LiquidWebWorker API for Horizontal Offloading of Stateless Computations. *Journal of Web Engineering*, 17(6):405–448, nov 2019.
[10] Sebastian Heil and Martin Gaedke. DCM: Dynamic Client-Server Code Migration. In Irene Garrigós, Juan Manuel Murillo Rodríguez, and Manuel Wimmer, editors, *Web Engineering*, pages 3–18, Cham, 2023. Springer Nature Switzerland.
[11] Sebastian Heil, Jan-Ingo Haas, and Martin Gaedke. Enhancing web applications with dynamic code migration capabilities. In Irene Garrigós, Juan Manuel Murillo Rodríguez, and Manuel Wimmer, editors, *Web Engineering*, pages 371–375, Cham, 2023. Springer Nature Switzerland.
[12] Jae-Yun Kim and Soo-Mook Moon. Disclosure: Efficient Instrumentation-Based Web App Migration for Liquid Computing. In *Web Engineering. Proc. of ICWE 2022*, pages 132–147, Cham, 2022. Springer.
[13] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proc. of IEEE INFOCOM*, pages 945–953. IEEE, 2012.
[14] Pradeep Kumar, Kakoli Banerjee, Niraj Singhal, Ajay Kumar, Sita Rani, Raman Kumar, and Cioca Adriana Lavinia. Verifiable, secure mobile agent migration in healthcare systems using a polynomial-based threshold secret sharing scheme with a blowfish algorithm. *Sensors*, 22(22), 2022.
[15] DB Lange and Mitsuru Oshima. Mobile agents with Java: the Aglet API. *World Wide Web*, 1:1–18, 1998.
[16] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. WebAssembly

Modules as Lightweight Containers for Liquid IoT Applications. In *Proc. of ICWE2021*, pages 328–336, Cham, 2021. Springer.
[17] Mohammed Maray and Junaid Shuja. Computation offloading in mobile cloud computing and mobile edge computing: Survey, taxonomy, and open issues. *Mobile Information Systems*, 2022:1121822, Jun 2022.
[18] Sehoon Park, Qichen Chen, and Heon Y. Yeom. PIOS: A platform-independent offloading system for a mobile web environment. *2013 IEEE 10th CCNC*, pages 137–142, 2013.
[19] Pauline P.L. Siu, N. Belaramani, C. L. Wang, and F. C.M. Lau. Context-aware state management for ubiquitous applications. In *Embedded and Ubiquitous Computing. EUC*, volume 3207, pages 776–785. Springer Verlag, 2004.
[20] Jari-pekka Voutilainen, Anna-liisa Mattila, Kari Systä, and Tommi Mikkonen. HTML5-based mobile agents for Web-of-Things. *Informatica*, 40(1):43–51, 2016.
[21] WebAssembly Community Group. WebAssembly Specification — WebAssembly 2.0 Draft 2022-12-15, 2022.

Biographies

Sebastian Heil is a postdoctoral researcher at the Distributed and Self-organizing Systems group of Chemnitz University of Technology, Germany. His research interests include Web Engineering, Automated Software Engineering, Design and Analysis of Web User Interfaces and Web of Things. He is a member of the ACM and has served as reviewer and PC member for several international journals and conferences including ACM TWEB, IJHCI, MONET, JWE, CHI, and as PC Chair of ICWE.

Lucas Schröder is a student assistant at the Distributed and Self-organizing Systems group of Chemnitz University of Technology, Germany. In his master studies and research he focuses on Knowledge Graphs, Data Quality, and Research Data Management.

Martin Gaedke is a full professor, director of the computing center, and dean of the Faculty of Computer Science at Chemnitz University of Technology. He has also served as a guest professor at TU Wien and lectured at several other universities. His research focuses on advancing collaboration for the hyper-connected society by contributing to the fields of Data & Web Engineering, Web of Things, Web Interaction, and smart trustworthy Collaboration. With more than 25 years of experience, and a Diploma and PhD from University of Karlsruhe (KIT), he has collaborated globally with leading companies and research institutions such as Daimler, Microsoft Research, and Hewlett-Packard, contributing to over 250 publications.