# JS-Distributor: decomposing monolith applications into microservices

Guilherme Salvador Escher
Computing Department, Federal University of São Carlos
São Carlos, SP, Brazil
guilhermeeschersalvador@gmail.com

Maurício Gallera de Almeida
Computing Department, Federal University of São Carlos
São Carlos, SP, Brazil
galmeidamauricio12@gmail.com

João Vitor Fidelis Cardozo
Computing Department, Federal University of São Carlos
São Carlos, SP, Brazil
joaov.cardozo@hotmail.com

Romeu Leite
Computing Department, Federal University of São Carlos
São Carlos, SP, Brazil
romeufilho@estudante.ufscar.br

Ivan Capeli Navas
Computing Department, Federal University of São Carlos
São Carlos, SP, Brazil
ivancn@estudante.ufscar.br

Vinicius Nordi Esperança
Computing Department, Federal University of São Carlos
São Carlos, SP, Brazil
viniciusnordiesperanca@gmail.com

Daniel Lucrédio
Computing Department, Federal University of São Carlos
São Carlos, SP, Brazil
daniel.lucredio@ufscar.br

## ABSTRACT

While the decomposition of monolithic applications into microservices has been widely studied, most existing approaches stop at architectural analysis or service identification, offering little support for the actual generation and deployment of microservices. This creates a gap between design and implementation, often requiring significant manual effort from developers. It also limits researchers, who lack practical tools to test and experiment with new decomposition strategies in real-world scenarios. To address these challenges, we present JS-Distributor, a tool that automates the transformation of monolithic JavaScript applications into microservices. JS-Distributor generates much of the required boilerplate code for Node.js, including server setup and inter-service communication code, supporting both HTTP APIs and asynchronous messaging systems. By bridging the gap between decomposition and deployment, the tool enables rapid experimentation with different microservice architectures. We evaluated JS-Distributor using a benchmark system commonly adopted in microservice research, successfully distributing the original monolith into functional microservices while preserving its original logic.

**Demo video:** https://doi.org/10.5281/zenodo.15477214

## KEYWORDS

Code generation, distributed applications, web development, web services, HTTP API, messaging systems

## 1 Introduction

Many monolithic applications are still in use and actively maintained today. While some could benefit from migrating to a microservice architecture, others are driven to it by the urgent need to overcome the limitations of monolithic systems [7]. Even modern service-based architectures may require adaptation, such as replacing REST-based communication with asynchronous messaging to better handle unexpected high-load scenarios [11]. The literature reports numerous efforts toward this migration [2, 3, 10, 12].

Among the many research gaps that need to be addressed, such as microservices identification an optimization, Abgaz et al. [2] highlight the lack of attention to deployment. In their 2023 systematic review, they found that most studies overlook the extraction and deployment phases, with only one explicitly addressing how to deploy the resulting microservices. They argue this is "undesirable because the ultimate test of microservice fitness will only arise once deployed." They also note the lack of tools to automate (or partially automate) the identification, extraction and deployment steps.

To support these last two steps, we developed JS-Distributor, a tool that automatically decomposes monolithic JavaScript code into independently deployable and executable microservices while preserving the original program logic. This work builds on previous research [6], which demonstrated the feasibility of this approach using a Java-based prototype with Remote Method Invocation. We extend the concept by adopting more flexible web-based technologies supported by Node.js and introducing asynchronous, message-based communication. By bridging the gap between research and practice, JS-Distributor assists industry practitioners in decomposing monolithic systems and enables researchers to empirically investigate runtime issues that only arise after deployment [2].

## 2 Illustrative scenario

Consider the following scenario: a developer needs to create a piece of software that performs the following functions:

(1) Generate an identifier by concatenating some user data (for example, the e-mail) with an RFC4122[1]-compatible UUID;
(2) Insert the user in the database, with the generated id as key;
(3) Fetch the recently-saved user from the database to retrieve the exact insertion timestamp; and

[1]https://www.ietf.org/rfc/rfc4122.txt

(4) Log the operation for auditing.

Initially, the developer could implement four functions running on a single machine using Node.js and Express[2]. However, as the system is deployed and the user base grows, performance delays may begin to appear. The team decides that it would be more efficient to split these functions into two HTTP-based microservices: (i) one responsible for UUID generation and logging, which makes sense as these are independent functions that do not require significant computational resources; and (ii) another for user creation and retrieval, where most of the heavy processing takes place, justifying the allocation of a dedicated server with greater memory capacity.

To implement this change, new HTTP endpoints must be created alongside the necessary code to handle communication between the newly separated services. Local function calls can remain unmodified, but any call targeting a service now hosted on a different server must be converted into a remote fetch operation. This involves marshalling the request parameters and responses according to the chosen communication format (such as JSON), handling asynchronous callbacks, common in languages like JavaScript, and managing new communication-related errors and exceptions.

And there is always room for improvement. For example, certain database read operations that do not require immediate consistency could be migrated to a separate server accessing a database replica that supports eventual consistency [4]. Additionally, the logging function could be integrated with a publish-subscribe messaging system, such as RabbitMQ[2], which operates asynchronously, easing the orchestration logic and reducing latency. Meanwhile, a client built with React[2] could make calls to the exposed microservices via HTTP API. Figure 1 illustrates these possibilities within a new architecture. Notice how the complexity increases alongside the effort required to implement these changes.

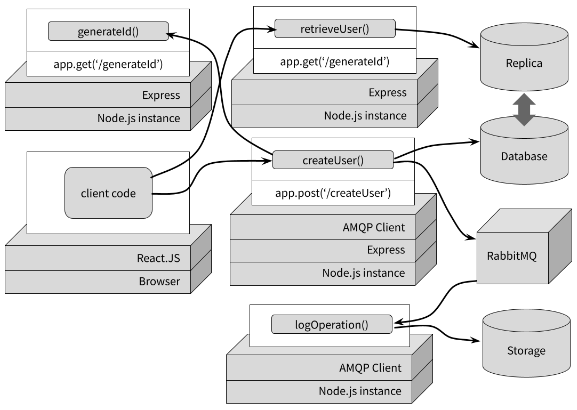


**Figure 1: Complex architecture with microservices.**

JS-Distributor was designed to facilitate these types of changes. It can automatically generate code to migrate standard JavaScript functions into distributed services that communicate via HTTP API or RabbitMQ messages. Additionally, it automatically detects when a function call has moved from a local server to a remote one and generates the necessary code to ensure correct operation. The tool replaces the original call with a remote invocation, handling the marshalling of request and response parameters transparently.

## 3 Usage

The process starts with a monolithic JavaScript application, such as the example from Listing 1.

```
1  import { v4 as uuidv4 } from 'uuid';
2  import { MongoClient } from 'mongodb';
3  function generateId(prefix) {
4    console.log(`Generating ID with prefix: ${prefix}`);
5    return `[${prefix}:${uuidv4()}]`;
6  }
7  async function createUser(email, name) {
8    console.log(`Create: email=${email}, name=${name}`);
9    const id = generateId(email);
10   const client = new MongoClient('mongodb://localhost
        :27017');
11   await client.connect();
12   await client.db('db').collection('Users').insertOne({
        _id: id, email, name });
13   console.log('User inserted successfully');
14   await client.close();
15 }
16 export default async function main() {
17   console.log(`Running application...`);
18   await createUser("user@email.com", "John Doe");
19   console.log(`Done!`);
20 }
```

**Listing 1: Sample JavaScript code.**

In the example from Listing 1, after the import statements (lines 1-2), there are three functions:

(1) `generateId()` (lines 3-6): generates an ID with some prefix;
(2) `createUser()` (lines 7-15): inserts a user in the database (MongoDB), first calling `generateId()`, passing the e-mail as a prefix; and
(3) `main()` (lines 16-20): runs the application by simply calling `createUser()` with an e-mail and name. This function is called in another script.

In order to distribute this code, the developer must provide a `config.yml` file such as the one shown in Listing 2.

```
servers:
  - id: alpha
    http:
      url: localhost
      port: 3000
    functions:
      - declarationPattern: generateId
        method: http-get
      - declarationPattern: createUser
        method: http-post
  - id: beta
    http:
      url: localhost
      port: 3001
    functions:
      - declarationPattern: main
        method: http-get
```

**Listing 2: JS-Distributor's configuration file.**

This configuration file completely describes how the existing functions must be distributed in different servers. In the example of Listing 2, there are two servers:

[2]nodejs.org, expressjs.com, www.rabbitmq.com, react.dev

(1) *alpha* (lines 2-10): runs on localhost, on port 3000. It will host functions `generateId()` (lines 7-8) through HTTP GET, and `createUser()` (lines 9-10) through HTTP POST; and
(2) *beta* (lines 11-17): runs on localhost, on port 3001. It will host function `main()` (lines 16-17) through HTTP GET.

Based on the input from Listings 1 and 2, JS-Distributor automatically generates startup code for the two servers. For example, for server *alpha* the code in Listing 3 is generated:

```
1  import express from 'express';
2  import {
3    generateId_localRef as generateId,
4    createUser_localRef as createUser
5  } from "./app.js";
6  const app = express();
7  const port = 3000;
8  app.use(express.json());
9  app.get('/generateId', (requestParameter,
       responseParameter) => {
10   const prefix = requestParameter.query.prefix;
11   const executionResult = generateId(prefix);
12   return responseParameter.json({
13     executionResult
14   });
15 });
16 app.post('/createUser', async (requestParameter,
       responseParameter) => {
17   const email = requestParameter.body.email;
18   const name = requestParameter.body.name;
19   const executionResult = await createUser(email,name);
20   return responseParameter.json({ executionResult });
21 });
22 app.listen(port, () => {
23   console.log('Server running in port ' + port);
24 });
```

**Listing 3: Code generated for the alpha server.**

Listing 3 contains, after the import statements (lines 1–5), the configuration of Express (lines 6–8). Next, the endpoint for each function configured to run on this server is generated according to the specified method: function `generateId()` is configured as a GET endpoint (lines 9–15), and function `createUser()` as a POST endpoint (lines 16–21). These code snippets also include the necessary statements for marshalling request and response parameters according to the chosen method. The original code for each function is imported from the original file, which is copied into each server. Notice, for example, how function `generateId()` is imported in line 3 and called in line 11. The same happens with `createUser()`, which is imported in line 4 and called in line 19. Finally, the server is started (lines 22–24).

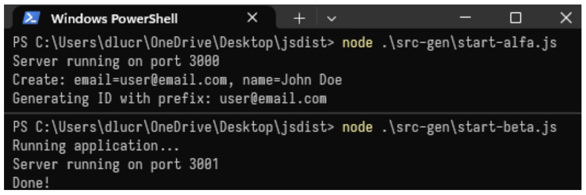
Due to space constraints, the code for the *beta* server is not shown here, but it is very similar to this one.

In addition to the server code, JS-Distributor also generates client code that can be used to connect to the functions hosted on that server. This code is inserted in all calls to functions that do not reside on the local server. Listing 4 shows an example of this code being generated in server *beta*, which does not host these functions and therefore must send requests to server *alpha*. Lines 1–5 contain the code for calling the `generateId()` function, and lines 6–15 for the `createUser()` function. Notice how each function correctly configures the HTTP method (GET or POST) and points to the address ("localhost") and port (3000) of the destination server. It also encapsulates the parameters as JSON in the body for POST requests or as key/value pairs in the query string for GET requests.

```
1  async function generateId(prefix) {
2    const response = await fetch(`http://localhost:3000/
        generateId?prefix=${prefix}`);
3    const { executionResult } = await response.json();
4    return executionResult;
5  }
6  async function createUser(email, name) {
7    const response = await fetch(`http://localhost:3000/
        createUser`, {
8      method: 'POST',
9      headers: { 'Content-Type': 'application/json' },
10     body: JSON.stringify({ "email": email,
11                            "name": name })
12   });
13   const { executionResult } = await response.json();
14   return executionResult;
15 }
```

**Listing 4: Code generated for the alpha functions.**

After running both servers, the output from Figure 2 is obtained. Notice how each server only produces the outputs for the functions it contains. Also, as desired, the overall global logic is maintained, with each individual microservice contributing towards it.



```
PS C:\Users\dlucr\OneDrive\Desktop\jsdist> node .\src-gen\start-alfa.js
Server running on port 3000
Create: email=user@email.com, name=John Doe
Generating ID with prefix: user@email.com
```

```
PS C:\Users\dlucr\OneDrive\Desktop\jsdist> node .\src-gen\start-beta.js
Running application...
Server running on port 3001
Done!
```

**Figure 2: Distributed execution in two servers.**

## 4 Internal design

Figure 3 shows the main components of JS-Distributor. Some were either reused or generated (shown in white), while others were developed for this research (shown in gray). The tool uses a JavaScript parser built with ANTLR and an open-source JavaScript grammar[3]. Based on this grammar (`lexer.g4` and `parser.g4`), ANTLR generates two components: a lexer/parser and a base visitor. The lexer/parser is the component that reads the source file and produces a parse tree. The base visitor is simply a set of functions that can be reused, by inheritance, to perform semantic actions upon visiting the nodes of the parse tree. We developed four visitors:

- PrepareTreeVisitor: visits the parse tree to establish parent-child relationships. This facilitates AST transformations, as ANTLR-generated parsers are not prepared for this task;
- ReplaceRemoteFunctionsVisitor: visits every function for every server. If a function belongs to a server, the visitor does nothing. Otherwise, it replaces the function body with a remote HTTP call or RabbitMQ message;
- FixAsyncFunctionsVisitor: checks if the transformed functions have the proper `async` prefix, necessary for remote calls and messages. It also propagates these prefixes up the call stack as needed;

[3]www.antlr.org, github.com/antlr/grammars-v4

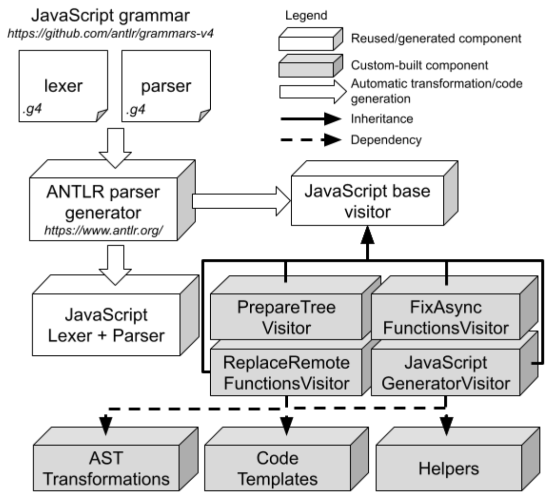

JavaScript grammar
https://github.com/antlr/grammars-v4
lexer .g4
parser .g4
Legend
Reused/generated component
Custom-built component
Automatic transformation/code generation
Inheritance
Dependency
ANTLR parser generator https://www.antlr.org/
JavaScript base visitor
JavaScript Lexer + Parser
PrepareTree Visitor
FixAsync FunctionsVisitor
ReplaceRemote FunctionsVisitor
JavaScript GeneratorVisitor
AST Transformations
Code Templates
Helpers

**Figure 3: Main components of the approach.**

- JavaScriptGeneratorVisitor: produces JavaScript code.

There are also other components that provide functions for AST transformations, templates for code generation and different helpers to suppot the job of the visitors.

Figure 4 illustrates how the components operate at runtime. The input is the monolithic source code and a configuration file (top-left corner). In this example, the monolith contains four functions (*f1*, *f2*, *f3*, *f4*), which are configured to be decomposed into two servers: *alpha* (*f1* and *f2*) and *beta* (*f3* and *f4*). The Lexer and Parser generate a parse tree for the monolith, which is then replicated, one copy for each server. In this process, the tool applies the necessary transformations by replacing, within the parse tree, the bodies of functions that do not belong to that server with remote call statements. As shown in the figure, in the parse tree for server *alpha*, the logic of functions *f3* and *f4* is replaced with calls to the beta server, where these functions are hosted. The same occurs in the *beta* server for functions *f1* and *f2*. The code on the right side illustrates the result. After code generation, each function is assigned to a specific server, while preserving the original program logic. The tool also generates startup code (not shown in the figure) to initialize the servers and enable function access.

## 5 Evaluation

To validate JS-Distributor, we used *Acme Air*, a benchmark system commonly adopted in microservices research [1, 2]. According to its original repository[4], it is a fictional airline application designed with key business requirements in mind, such as the ability to scale to billions of web API calls per day, support for deployment in public clouds, and the need to handle multiple user interaction channels. *Acme Air* has implementations in both Java and JavaScript; we used the JavaScript version for our purposes. Since the project had not been updated for a long time, we modernized the code: we upgraded library versions, adapted module import/export statements to follow modern standards, and replaced callback-based logic with proper async/await equivalents. The modernized version is available as a fork of the original repository[5].

In our experiments, we successfully decomposed *Acme Air* using various combinations of communication technologies, including HTTP (GET and POST) and asynchronous messaging via RabbitMQ. In all scenarios, the resulting microservices preserved the exact same functionality as the original monolith. For example, Figure 5 shows the system running on four servers. The front end is displayed on the right, while the four back-end services, running as Docker containers named *alpha*, *beta*, *gamma*, and *delta*, communicate through HTTP and RabbitMQ, demonstrating coordinated execution. We also successfully deployed *Acme Air* across ten different servers, as shown in Figure 6. Achieving such a highly complex scenario, combining multiple communication technologies and coordinating across several servers, without modifying a single line of code, demonstrates the tool's reliability and practical value.

We also tested JS-Distributor with an application developed in React Native[6]. The application, which was developed in our research group[7], serves for members of our university community to request and offer rides during their daily commutes. Figure 7 illustrates the architecture and a screenshot of the application. It uses cloud-based Firebase and Google APIs for its functions, as depicted in the figure. In the mobile device, there are functions for geolocation and storage. Highlighted in the figure there is a function for e-mail validation, which is used to restrict access to users that belong to the university's academic community, based on their e-mail address domain. We used JS-Distributor to automatically migrate this function, which originally was running on the mobile device, to a separate server. We tested two configurations, using HTTP POST and RabbitMQ, and both worked correctly.

These and other tests were conducted in a simulated distributed environment, using a single machine, and in a laboratory setting with multiple computers. In all cases, the code executed correctly.

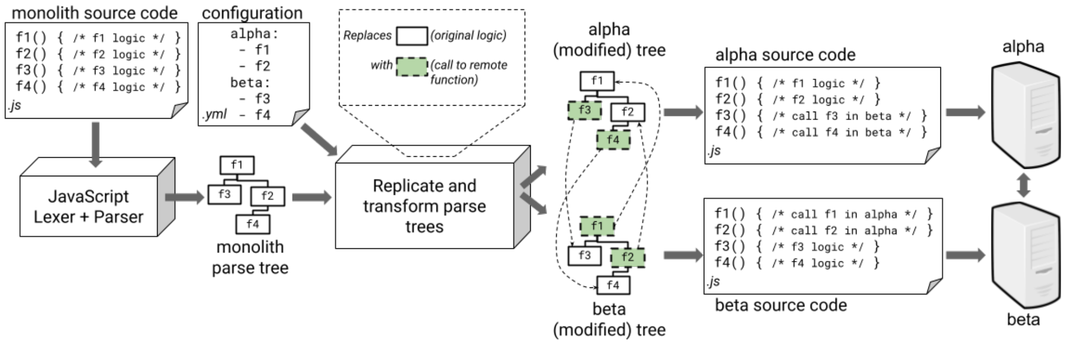
## 6 Related tools

Abgaz et al. [2] conducted an extensive systematic literature review on the decomposition of monolith applications into microservices architectures, presenting their findings as a framework called M2MDF. In addition to mapping existing research, the authors also highlight research gaps and opportunities in each part of their framework, emphasizing that only one study addresses the deployment of extracted microservices, which is a critical limitation, since true validation of microservice fitness occurs only after deployment [2]. This is precisely the focus of our research: enabling testing and experimentation in the actual deployment environment.

The study mentioned by Abgaz et al. [2] is the one conducted by Abdullah et al. [1]. It explores the actual deployment of different microservice configurations to measure response time, throughput and CPU usage. Their approach analyzes runtime logs to identify URL partitions with similar behavior, generating microservices accordingly. Results show that carefully chosen microservices improve performance and scalability. However, instead of truly partitioning the monolith, their solution duplicates it and redirects calls by URL.

[4]github.com/acmeair

[5]github.com/dlucredio/acmeair-nodejs
[6]reactnative.dev
[7]github.com/joaofidelisc/caronasuniversitarias



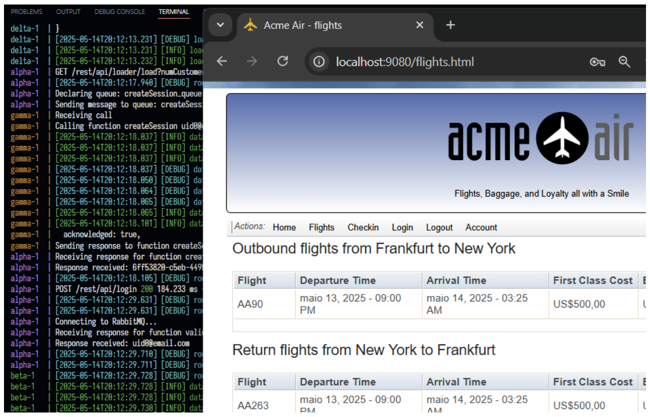
```
f1() { /* f1 logic */ }
f2() { /* f2 logic */ }
f3() { /* f3 logic */ }
f4() { /* f4 logic */ }
.js
```

```
alpha:
  - f1
  - f2
beta:
  - f3
.yml - f4
```

```
f1() { /* f1 logic */ }
f2() { /* f2 logic */ }
f3() { /* call f3 in beta */ }
f4() { /* call f4 in beta */ }
.js
```

```
f1() { /* call f1 in alpha */ }
f2() { /* call f2 in alpha */ }
f3() { /* f3 logic */ }
f4() { /* f4 logic */ }
.js
```

**Figure 4: Internal runtime details.**



```
delta-1  | }
delta-1  | [2025-05-14T20:12:13.231] [DEBUG] loa
delta-1  | [2025-05-14T20:12:13.231] [INFO] loa
delta-1  | [2025-05-14T20:12:13.232] [INFO] loa
alpha-1  | GET /rest/api/loader/load?numCustome
alpha-1  | [2025-05-14T20:12:17.940] [DEBUG] ro
alpha-1  | Declaring queue: createSession_queue
alpha-1  | Sending message to queue: createSess
gamma-1  | Receiving call
gamma-1  | Calling function createSession uid0@
gamma-1  | [2025-05-14T20:12:18.037] [INFO] dat
gamma-1  | [2025-05-14T20:12:18.037] [INFO] dat
gamma-1  | [2025-05-14T20:12:18.037] [INFO] dat
gamma-1  | [2025-05-14T20:12:18.037] [DEBUG] da
gamma-1  | [2025-05-14T20:12:18.050] [DEBUG] da
gamma-1  | [2025-05-14T20:12:18.064] [DEBUG] da
gamma-1  | [2025-05-14T20:12:18.065] [DEBUG] da
gamma-1  | [2025-05-14T20:12:18.065] [INFO] dat
gamma-1  | [2025-05-14T20:12:18.101] [INFO] dat
gamma-1  |   acknowledged: true,
gamma-1  | Sending response to function createS
alpha-1  | Receiving response for function crea
alpha-1  | Response received: 6ff53820-c5eb-449
alpha-1  | [2025-05-14T20:12:18.105] [DEBUG] ro
alpha-1  | POST /rest/api/login 200 184.233 ms
alpha-1  | [2025-05-14T20:12:29.631] [DEBUG] ro
alpha-1  | [2025-05-14T20:12:29.631] [DEBUG] ro
alpha-1  | Connecting to RabbitMQ...
alpha-1  | Receiving response for function vali
alpha-1  | Response received: uid0@email.com
alpha-1  | [2025-05-14T20:12:29.710] [DEBUG] ro
alpha-1  | [2025-05-14T20:12:29.711] [DEBUG] ro
beta-1   | [2025-05-14T20:12:29.728] [DEBUG] ro
beta-1   | [2025-05-14T20:12:29.728] [INFO] dat
beta-1   | [2025-05-14T20:12:29.728] [INFO] dat
beta-1   | [2025-05-14T20:12:29.730] [INFO] dat
```

Outbound flights from Frankfurt to New York

| Flight | Departure Time | Arrival Time | First Class Cost |
|---|---|---|---|
| AA90 | maio 13, 2025 - 09:00 PM | maio 14, 2025 - 03:25 AM | US$500,00 |

Return flights from New York to Frankfurt

| Flight | Departure Time | Arrival Time | First Class Cost |
|---|---|---|---|
| AA263 | maio 13, 2025 - 09:00 | maio 14, 2025 - 03:25 | US$500,00 |

**Figure 5: *Acme Air* system running on four servers.**

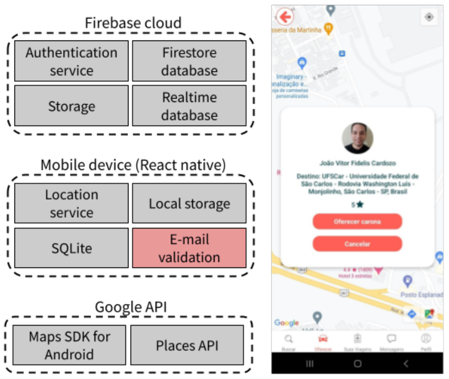


**Figure 7: Mobile application used in the evaluation.**

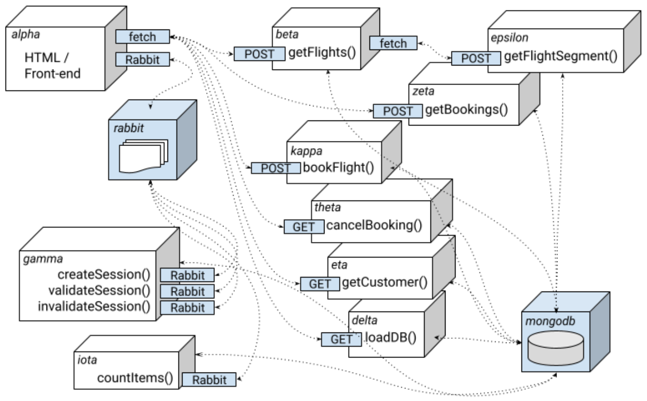


**Figure 6: Ten-server configuration successfully tested.**

While effective, this leaves room for improvement — the code is never actually split, which is what we propose.

A related study from our research group [6] implemented a partitioning algorithm that, unlike previous approaches, does not duplicate the monolith but instead generates actual code partitions. The prototype was Java-based and relied on Remote Method Invocation for communication. The experiments were conducted using Apache Tomcat, which is arguably better suited for monolithic architectures. Nevertheless, it served as an excellent proof of concept for the viability of the approach. As a result, response times were worse in the microservice-based version, but improvements in memory usage were still observed. Additionally, the study relied on manual microservice identification rather than more automated or optimized techniques, such as those presented elsewhere [1, 2]. In contrast, JS-Distributor follows a similar direction but adopts more flexible, web-based technologies (JavaScript), and includes support for asynchronous, message-based communication.

Kaplunovich [9] presents a fully automatic approach to decomposing monoliths, targeting a serverless architecture instead of microservices. Each Java method is transformed into an AWS Lambda-compatible Node.js function. While the approach faces limitations with polymorphism, hierarchies and enumerations, the transformation itself shows promising results. However, the paper does not evaluate performance against the original monolith, nor does it allow developers to choose which functions run locally or remotely.

Carvalho and Araújo [5] follow a similar path, proposing the conversion of Node.js applications into serverless functions without analyzing which should run remotely or locally. Their experiments showed that some scenarios do not benefit from decomposition, confirming findings by Esperança and Lucrédio [6], though some cases saw performance gains. As future work, they suggest automatic inspection of the application and giving developers control over which functions to migrate, directions we pursue in this paper.

Industry offers tools with similar purposes. Service Weaver[8] enables automatic code distribution based on components (deployable services). Requests to the Weaver are routed to the appropriate component without requiring the client to know its location. However, developers must explicitly use the Weaver to invoke services, whereas in JS-Distributor, any regular function call can be automatically transformed into a remote service invocation. Additionally, Service Weaver supports only Go, limiting its applicability to front-end development. JS-Distributor is also confined to a single programming language (JavaScript), but one that is widely used in front-end environments, as shown in our evaluation.

Cadence and Temporal[9] offer solutions that not only handle distribution but also include features like persistence and fault tolerance. Despite these additional capabilities, their main value lies in abstracting distributed execution and orchestration for the developer. However, they still require services to be explicitly defined before use, unlike our approach, which lets developers postpone or modify this decision with minimal effort. Additionally, these platforms require full adoption, whereas our method produces standalone JavaScript code. This means developers can discontinue the use of the generator and continue development with the generated code as if it had been manually crafted.

Another tool with a similar purpose is ChatGPT[10] and other LLMs (Large Language Models). An LLM can split monolithic code into distributed services across various languages and frameworks, often producing functionally equivalent code. Their advantage lies in broad language and technology support. However, being based on nondeterministic processes, results can vary and contain errors, making human review essential. In contrast, our approach uses a parser defined by an EBNF grammar, ensuring deterministic and consistent code generation that developers can more reliably trust.

An AI-based solution is also part of Mono2Micro[11] [8], a tool that focuses on the identification of optimal microservice candidates. However, as highlighted by Abgaz et al. [2], manual effort is necessary to actually create and deploy the microservices.

As it can be seen, the approach we propose in this paper fulfills some gaps reported in the literature, and has the potential to leverage further research, specially in terms of experiments involving deployment in production environments.

## 7 Conclusion

In this paper, we present JS-Distributor, a tool designed to facilitate the migration of monolithic applications to a microservice architecture. It automatically generates code for server configuration and communication, supporting both HTTP API requests and asynchronous messaging via RabbitMQ queues. With JS-Distributor, developers can decompose a monolith into microservices with minimal manual effort, as the tool handles much of the complexity of distribution and inter-service communication through code generation. It addresses an important research gap in monolith decomposition [2] by supporting both the extraction and deployment of services.

From a practitioner's perspective, JS-Distributor enables a more agile and experimental migration process. Developers can test different deployment options by simply updating a configuration file and regenerating the code. They can move functions between servers, switch communication methods (e.g., from GET to POST), or adopt asynchronous messaging, all without manually rewriting code.

From a researcher's perspective, JS-Distributor bridges the gap between microservices research and practical application. By automating much of the effort involved in separating and deploying services, experimenting with different configurations and architectures becomes nearly effortless—often as simple as pressing a button. This facilitates the collection of empirical data beyond the theoretical metrics commonly used in academic studies. In future work, we plan to conduct such empirical evaluations to put JS-Distributor into action in the scenarios it was designed to support.

Currently, JS-Distributor supports HTTP API GET and POST, RabbitMQ queues (including broadcasting), and JSON data format. Future work aims to extend support to more advanced messaging scenarios, such as multiple worker queues, message replication, and load balancing. Integration with common cloud-native technologies like CI/CD pipelines and Kubernetes will also be investigated.

One practical limitation of the tool is its support for a single programming language. Although JavaScript is one of the most widely used languages for web development, on both the front and back end, polyglot environments are a key advantage of microservices and should not be overlooked. Future work could explore additional languages and improved interoperability. Based on our experience, porting JS-Distributor to other languages may be challenging, especially in statically typed ones, as the lack of type checking in JavaScript greatly simplifies the underlying transformations. In this context, supporting TypeScript would be a natural first step, laying the groundwork for broader multi-language support.

We are also developing integrated features for testing and debugging distributed architectures, enabling developers to systematically validate behavior across different deployment configurations [2]. In addition, we plan to enhance the tool to support database partitioning, allowing each microservice to access a distinct portion of the data. Finally, since our tool operates on JavaScript, it opens the possibility for investigations in the area of Micro Front-Ends.

## ARTIFACT AVAILABILITY

The tool is available as open source software under the **MIT License** in the following address:

https://doi.org/10.5281/zenodo.15880232

## ACKNOWLEDGMENTS

Funding: This work was partially supported by São Paulo Research Foundation (FAPESP), grant #2021/06983-2.

We would like to thank the reviewers for their insightful suggestions and ideas, which helped improve the paper.

## REFERENCES

[1] Muhammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. 2019. Unsupervised learning approach for web application auto-decomposition into microservices. *Journal of Systems and Software* 151 (2019), 243–257. doi:10.1016/j.jss.2019.02.031

[8] serviceweaver.dev
[9] cadenceworkflow.io, temporal.io
[10] openai.com/blog/chatgpt
[11] www.ibm.com/blog/announcement/ibm-mono2micro

[2] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. 2023. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4213–4242. doi:10.1109/TSE.2023.3287297

[3] Marco Autili, Gianluca Filippone, and Massimo Tivoli. 2023. Migrating from Monoliths to Microservices: Enforcing Correct Coordination. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 113–118. doi:10.1109/ASEW60602.2023.00020

[4] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (2012), 23–29. doi:10.1109/MC.2012.37

[5] Leonardo Rebouças de Carvalho and Aletéia Patrícia Favacho de Araújo. 2019. Framework Node2FaaS: Automatic NodeJS Application Converter for Function as a Service. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science* (Heraklion, Crete, Greece) *(CLOSER 2019)*. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, 271–278. doi:10.5220/0007677902710278

[6] Vinicius Nordi Esperança and Daniel Lucrédio. 2017. Late Decomposition of Applications into Services through Model-Driven Engineering. In *Proceedings of the XXXI Brazilian Symposium on Software Engineering* (Fortaleza, CE, Brazil) *(SBES '17)*. Association for Computing Machinery, New York, NY, USA, 164–173. doi:10.1145/3131151.3131165

[7] Hassan Farsi, Driss Allaki, Abdeslam En-nouaary, and Mohamed Dahchour. 2023. Dealing with Anti-Patterns When Migrating from Monoliths to Microservices: Challenges and Research Directions. In *2023 IEEE 6th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech)*. 1–8. doi:10.1109/CloudTech58737.2023.10366131

[8] Anup K. Kalia, Jin Xiao, Chen Lin, Saurabh Sinha, John Rofrano, Maja Vukovic, and Debasish Banerjee. 2020. Mono2Micro: an AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1606–1610. doi:10.1145/3368089.3417933

[9] Alex Kaplunovich. 2019. ToLambda: automatic path to serverless architectures. In *Proceedings of the 3rd International Workshop on Refactoring* (Montreal, Quebec, Canada) *(IWOR '19)*. IEEE Press, 1–8. doi:10.1109/IWoR.2019.00008

[10] Idris Oumoussa and Rajaa Saidi. 2024. Evolution of Microservices Identification in Monolith Decomposition: A Systematic Review. *IEEE Access* 12 (2024), 23389–23405. doi:10.1109/ACCESS.2024.3365079

[11] Sunil Kumar A R. 2023. REST vs. Messaging for Microservices. *Informatics - An eGovernance Publication from national Informatics Centre* (2023). https://informatics.nic.in/uploads/pdfs/19c88a8a_informatics_july_2023.pdf

[12] Yamina Romani, Okba Tibermacine, and Chouki Tibermacine. 2022. Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 15–19. doi:10.1109/ICSA-C54293.2022.00010