Js-Distributor Extension for Automated Test Migration From Monoliths to Microservices

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

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

Abstract

Despite the growing adoption of microservices, many monolithic systems remain in production due to the high cost and risk of migration. While existing approaches support various phases of this transition, there is still a significant gap in the deployment phase, especially regarding the automated migration of test cases. This limitation introduces challenges such as broken tests due to cross-service dependencies and the need to validate newly introduced remote communications. In this paper, we propose an extension to the JS-Distributor tool to address these issues. Our approach automates the migration of test cases from monolithic systems to microservice-based architectures, generates double functions to enable isolated unit testing, and creates tests for validating remote interactions between services. The solution is driven by a YAML-based configuration that defines the distribution of functionalities across services. We evaluate the proposed approach by demonstrating that migrated tests can be executed both independently within microservices and in a fully deployed distributed environment. Results show that the generated tests are capable of detecting both pre-existing bugs and new issues introduced during migration, thus helping preserve system functionality and improving reliability in the transition process.

Demo video: https://doi.org/10.5281/zenodo.20369693

Keywords

Code generation, distributed applications, decomposition, test case, integration test case, migration validation

1 Introduction

Despite the industry shift toward distributed systems, many large-scale monolithic applications remain in production [10]. This persistence is often due to the considerable time and risk associated with migration. As Abgaz et al. [2] illustrate, a substantial body of literature addresses the identification of microservice boundaries through manual, semi-automated, and fully automated approaches. These methodologies employ diverse strategies, such as semantic proximity based on API specifications [3], machine learning techniques [7] and clustering methods [9]. However, there remains a significant research gap in deployment, as pointed out by Abdullah et al. [1]. While their work advances the literature toward this direction, focusing on optimizing decomposition for performance and cost, it leaves a critical gap: the automated migration of test cases.

This omission is problematic because it leaves the behavior of cross-boundary test function calls to the architect's discretion. If a test invokes a function outside its designated server in the new microservice-based architecture, it can no longer run without starting the target server to respond to the remote invocation, breaking unit tests that run on Continuous Integration/Continuous Deployment (CI/CD) pipelines. Furthermore, the introduction of new remote calls for the microservices to communicate leads to the need for new tests, to verify and validate if parameters and return values are treated correctly, according to the chosen communication technology. Section 2 illustrates these problems.

This highlights the need for a tool to:

- Migrate the test cases from the monolith to the microservices so that they can run in the new distributed architecture;
- Generate double functions so that unit tests can execute in individual microservices without the need to run multiple servers; and
- Test the newly generated remote functions in order to validate parameters and return values.

In light of these requirements, we propose an extension to JS-Distributor, a tool that helps in the migration of monoliths to microservices with focus on producing executable and deployable code [5]. JS-Distributor automates the deployment phase using a single YAML configuration file, which dictates the specific server allocation for each extracted functionality. In this paper, we describe how the extension fulfills the stated requirements. We also present an evaluation that demonstrates how the migrated tests can be executed either individually, as unit tests within each microservice, or in the complete environment, with multiple running servers. We also demonstrate that the generated tests can detect the same bugs as those found in the monolith, as well as new bugs introduced in the migration process, helping ensure that the original monolith's functionality is preserved in the new microservice-based version.

2 Illustrative Scenario

Consider a typical migration scenario: A software architect is tasked with decomposing a monolithic application into microservices. Because this process is highly error-prone, a requirement is that the existing unit and integration tests must remain fully operational post-migration to ensure functional equivalence. Listing 1 presents an example of monolithic functions targeted for migration. Notice that the function `saveCustomer` invokes `toUpperCase` and `split`.

```
1 export function saveCustomer(name, address, ...) {
2     const words = split(name);
3     const lastName = toUpperCase(words.at(-1));
4     const middleNames = words.slice(1,-1);
5     // save customer in the database
6     ...
7 }
8 export function toUpperCase(str){//Converts to upper
      case}
```

```
9 export function split(str,separator=","){//Splits a
      string}
```

Listing 1: Monolith Code Example

Listing 2 shows a possible test suite for these functions, using Jest[1] and Supertest[2]. Lines 1–13 group unit tests for the string manipulation functions (specifically, lines 2–7 test the `split` function and lines 8–12 test the `toUpperCase` function). Lines 14–20 focus on database-related functionality. In the monolith, these test cases can run normally, as each function can be invoked locally.

Suppose that the monolith is partitioned into two servers: alpha and beta. If the `saveCustomer` function is deployed to the alpha server, while its dependencies, `split` and `toUpperCase`, reside on the beta server, the `saveCustomer` test, depicted in Listing 2, will not execute unless both servers are up and running, because it invokes functions that do not exist on the alpha server. This interdependency affects CI/CD automation. Furthermore, assume that `split` is exposed as an HTTP endpoint and `toUpperCase` as an Advanced Message Queuing Protocol (AMQP) consumer. The system then becomes susceptible to bugs introduced by these new layers, highlighting the need for additional testing.

Manually migrating these tests is tedious and time-consuming. Engineers must rewrite existing tests for each new microservice to validate their isolated functionality. Furthermore, they must develop custom mechanisms to execute inter-service calls, which is essential for surfacing communication-related bugs.

```
1 describe('String Functions', () => {
2  test('should split a string by separator', () => {
3   const str = "hello,world";
4   const separator = ",";
5   const result = split(str, separator);
6   expect(result).toEqual(["hello", "world"]);
7  });
8  test('should change to upper case', () => {
9   const str = "hello world";
10  const result = toUpperCase(str);
11  expect(result).toBe("HELLO WORLD");
12 });
13 })
14 describe('Use Database',  () => {
15  test('Should return customer', async () => {
16   const in = { name: "Lumi", addr: "R. Flores",...};
17   const out = { name: "Lumi", addr: "R. Flores", ...};
18   const res = await saveCustomer(in.name,in.addr, ...);
19   expect(res).toEqual(out);
20 });})
```

Listing 2: Monolith Test Case Example

3 Functionalities

We now detail how JS-Distributor is utilized to correctly migrate both the monolithic functions and their associated test cases to the distributed servers. Because this paper serves as an extension of prior work [5], we focus on the test migration process, which is the primary contribution of this study.

3.1 Test Case Isolation

The baseline tool is already capable of migrating monolith code, including the test cases, to microservices [5]. Therefore, basic end-to-end test scenarios, where test cases execute while all servers are running, were already supported.

However, in order to execute unit tests, where each microservice is tested individually, potentially within a CI/CD pipeline, the generated code does not work. This is because JS-Distributor replaces local calls with remote calls, even in the test cases.

In this extension, we solve this problem by cloning the source code of each migrated microservice, including all configuration files and other assets, into new folders dedicated to independent unit testing. These folders will receive the *-test-server* suffix. The copies are then modified to replace every remote call with a local substitute, so that they can run locally. Except for these remote calls, the original functionality is retained.

As a result, there are now two options for testing the new architecture. The original monolith tests, which potentially have interdependencies between multiple microservices, can be executed as integration tests that exercise the entire distributed architecture. And the tests in the cloned microservices can be executed as unit tests, exercising the logic within themselves.

The next section describes how these local substitutes are defined.

3.2 Test Mocking Techniques

Prior to migration, it is highly recommended that the software architect review the existing test scenarios to determine the appropriate isolation strategy. For example, the architect could identify which dependencies require mocking and which should be directly copied to these newly cloned test servers. The goal is to ensure that microservice test cases can be executed independently.

To support the first of these strategies, the tool enables the creation of custom double functions that eliminate the need for external network calls and allow the developer to focus strictly on validating internal functionality within the isolated test server. To utilize this, the developer declares the double function, implements the desired logic, and names it by prepending *mock_* to the original function's name. JS-Distributor then automatically replaces the original function with this mocked implementation on any test server where the **original** function is **not deployed**.

If a double function is not provided, JS-Distributor copies the original function that has been moved to a different server and uses it as a local double. This option frees the developer from having to define double functions, but it introduces code duplication across different microservices. This redundancy poses a risk as the system evolves, potentially leading to divergent changes or architectural drift where outdated logic persists in one service while being updated in another. Consequently, this approach is highly discouraged for large components or functions containing critical business logic.

3.3 Testing the Remote Calls

The proposed extension enables the detection of communication-related bugs, as well as request and response parsing issues, directly

[1] jestjs.io
[2] github.com/forwardemail/supertest

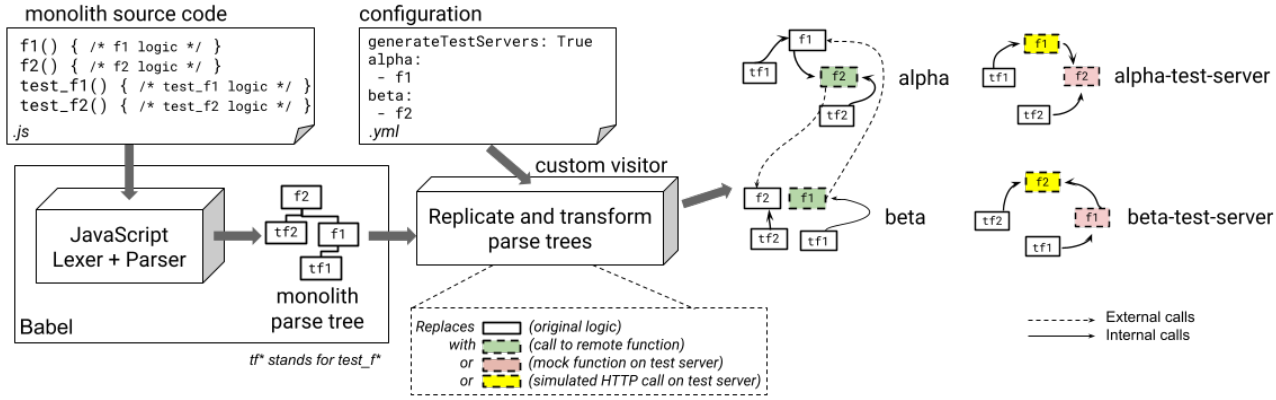

Figure 1: JS-Distributor internal design with Babel

on the test server. This is achieved by detecting whether a test invokes a remote function and automatically replacing the original function call with a simulated HTTP request generated using Supertest. Listing 3 shows the generated `saveCustomerApiTest` function. This function makes a simulated request to the GET endpoint, passing all parameters into the URL query string, and parses the resulting HTTP response as a JSON string.

```
1 async function saveCustomerApiTest(name, addr, ...) {
2     return JSON.parse((await request(app).get(`/
       saveCustomer?name=${name}&addr=${addr}...`)).text)
       .executionResult;
3 }
```

Listing 3: Generated HTTP Mocked Call

This feature allows integration-level validation tests to run exclusively on the current server, enabling the validation of parameters and return values without the need to run all servers.

JS-Distributor natively supports AMQP calls, but this feature was not implemented in the current extension. Therefore, adding this capability is left for future work.

4 Internal design

Under the hood, JS-Distributor uses Babel[3] for Abstract Syntax Tree (AST) code manipulation, representing an evolution from the previous version of JS-Distributor, which used ANTLR[4] [5]. Babel now handles the parsing of the monolithic codebase and provides a native visitor pattern implementation. Consequently, this eliminates the need to maintain a custom grammar and lexer.

Figure 1 illustrates how the extended JS-Distributor tool works. A custom Babel visitor traverses the monolith's ASTs and replicates the original functions, replacing local calls with remote ones (green boxes, in the figure), so that the distributed version's functionality is equivalent to the monolith's. In the test servers, these same functions are replaced with local doubles (red boxes, in the figure). And those functions that are exposed as HTTP endpoints are tested using simulated HTTP requests in Supertest (yellow boxes, in the figure). The result is that each test server can run independently (notice that there are no arrows representing external calls in the test servers).

5 Usage

In JS-Distributor, the integration and deployment phase is orchestrated through the following steps:

5.1 Declaring the module

A declarative YAML configuration file maps specific functions to their target servers and defines essential auxiliary parameters, such as the source code directory of the JavaScript monolith and the output storage location. In the proposed extension, a new optional boolean configuration named **generateTestServers** was introduced. This instructs JS-Distributor to generate the test servers, including the double functions and simulated HTTP request tests.

An example of this configuration is shown in Listing 4. Lines 1–4 contain the global configuration, with line 3 showing the new flag. The subsequent lines (5–22) declare each server's name, connection settings, and function settings, such as the name of the function residing on each server and the communication method to be used. In this example, `saveCustomer` is deployed to alpha (as an HTTP POST endpoint), and `split` and `toUpperCase` are deployed to beta (as HTTP GET and POST endpoints).

```
1  codeGenerationParameters:
2    ...
3    generateTestServers: true  # New flag
4  servers:
5    - id: alpha
6      http:
7        url: localhost
8        port: 3000
9      ...
10     functions:
11       - declarationPattern: 'saveCustomer'
12         method: http-post
13   - id: beta
14     http:
15       url: localhost
16       port: 3001
17     functions:
18       - declarationPattern: split
19         method: http-get
```

[3] babeljs.io
[4] www.antlr.org

```
20        - declarationPattern: 'toUpperCase'
21          method: http-post
```

Listing 4: Yaml Configuration File

5.2 Execution and Test Transformation

Once the input codebase and YAML configuration are ready, JS-Distributor is executed to initiate the monolithic decomposition. During this phase, the existing test suite is automatically partitioned and transformed, following the procedure explained in the previous sections. In this example, four folders are generated: `alpha`, `alpha-test-server`, `beta` and `beta-test-server`.

5.3 Execution of Unit Test Cases

Once the isolated test servers are generated, the developer can navigate to the respective output directories for the test servers and execute the test suite using a standard package manager (e.g., via the `npm`[5] `test` command). This can also be done in a CI/CD pipeline, as there are no external dependencies. This step validates the internal logic of each isolated component, as well as the newly introduced communication layer.

5.4 Execution of Integration Tests

Finally, by concurrently initializing all generated servers, the developer can execute the transformed test scenarios as full integration tests. This phase is critical for validating inter-service dependencies, ensuring that remote invocations are working correctly, such as a function on alpha server invoking a function on the beta server through HTTP.

6 Evaluation

In this evaluation, we defined five criteria:

- **Independent test execution:** It should be possible to run isolated unit tests that exercise each microservice's own logic. This should be possible without starting any server;
- **Independent endpoint testing:** It should be possible to test the newly generated HTTP endpoints without the need to fire up the servers;
- **Integrated test execution:** It should be possible to run the original monolith tests in order to exercise the entire system's logic as it was in the monolith. This requires all servers to be up and running;
- **Ability to detect the same bugs:** After migration, it should be possible to detect the same bugs as before;
- **Ability to detect new bugs:** Bugs introduced during the migration process should be detected; and
- **Preservation of test coverage:** After migration, the test suite should achieve a level of code coverage comparable to that of the original monolith, ensuring that no significant portions of the system become untested.

For the evaluation, it was used *AcmeAir*[6], a well-established benchmark application frequently cited in microservice literature [2, 7]. A modernized version was used, where all runtime libraries were updated[7].

However, *AcmeAir* did not include a test suite. Therefore, *Claude Code*[8] (running the *Opus 4.7 medium effort* model) was used to generate test cases. With the objective of generating tests that could run on the monolith, detect bugs, and present a high level of coverage, the following prompt was provided (in Portuguese):

"I need unit tests created for all functions in route/index.js and integration tests that verify the combined behavior of these functions. I'm looking for high test coverage."

Listing 5 illustrates one generated test case. Lines 1–2 show the test scenario setup, where a simulated database interruption is expected. Line 3 shows the actual request being made, and line 5 expects an exception to be thrown. This test fails if an exception is not properly handled and formatted before being returned to the remote caller.

```
1 it('Propagates the exception when validateCustomer (
      findOne) throws - call out side try/catch block',
      async () => {
2     spies.findOne.mockRejectedValueOnce(new Error('db
      down'));
3     const req = makeReq({ body: { login: 'alice',
      password: 'pwd' } });
4     const res = makeRes();
5     await expect(routes.login(req, res)).rejects.
      toThrow('db down');
6     expect(res.cookie).toHaveBeenCalledWith('sessionid
      ', '');
7 });
```

Listing 5: AcmeAir Test Example

A total of fifty-one (51) unit and integration test suites were generated. All test cases were reviewed by the authors, and passed successfully when executed in the monolith. Because we wanted to verify if the migration process does not change the tests' ability to detect bugs, we manually introduced a mutation by removing a simple verification in the monolith's code. As a result, one test case failed, as shown in Figure 2.

Next, JS-Distributor was used to split *AcmeAir* into two servers, alpha and beta. Some functions were deployed to alpha, some to beta, and some functions were maintained in both servers (this is possible in JS-Distributor, and especially useful for some helper functions used in many places).

Since the new configuration flag was set to true, the corresponding test environments, alpha-test-server and beta-test-server, were also automatically generated.

The migrated tests ran on the distributed version. In order to allow inter-service communication to succeed, alpha and beta servers were started. The bug detected in the monolith was also detected by the migrated tests in the microservice-based architecture.

In the following step the test cases within alpha-test-server and beta-test-server were executed. As expected, the tests were successfully executed in isolation, i.e. without requiring the servers to be started. The bug detected in the monolith was also detected here, in both test servers, because the bug was contained in a helper function that was present in both servers.

[5] www.npmjs.com
[6] github.com/acmeair
[7] github.com/dlucredio/acmeair-nodejs
[8] claude.com/product/claude-code

Figure 2: Test results for the *AcmeAir* monolith.

However, the test cases in *alpha-test-server* revealed that a new integration error had been introduced during the decomposition process. Because `validateCustomer` was transformed into an HTTP POST endpoint, any exception raised inside it (for example, when the database is down) is not automatically encapsulated into a proper HTTP response for the caller to process (JS-Distributor does not implement this encapsulation). Instead, it is captured by the API HTTP layer (express[9]) and sent to the caller in the form of HTML content. This causes another exception when the client code tries to parse this content as JSON.

Lines 1–3 of Listing 6 show the generated simulated HTTP POST endpoint for `validateCustomerApiTest`. It is possible to notice that there is no try/catch block surrounding this endpoint.

```
1 export async function validateCustomerApiTest(username,
      password) {
2   return JSON.parse((await request(app).post('/
     validateCustomer').send(
3   {"username": username, "password": password})).text
     ).executionResult;}
```

Listing 6: Generated code sample on alpha-test-server

Without an error-handling mechanism at this newly established network boundary, a parsing error ensues, thus preventing the caller from receiving the anticipated error payload. The resulting error output is depicted in Figure 3, which shows the manually introduced bug already detected in the monolith and the newly introduced communication error.

We then proceeded to analyze test case coverage. Table 1 summarizes the results.

As can be seen, statement coverage in the distributed version is very similar to that of the monolith. The lower coverage observed in the beta integration tests can be explained by the fact that one function allocated to the beta server is never executed, since it is only invoked by a function located in the alpha server. As a result,

[9]expressjs.com

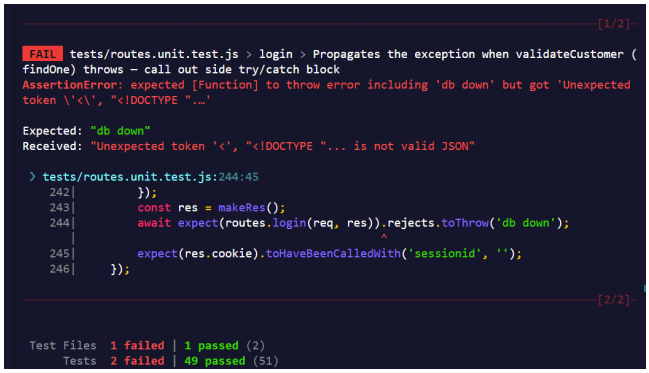

Figure 3: Test Results for *alpha-test-server*

Table 1: Coverage analysis.

	Monolith	Distributed			
		Unit		Integration	
		alpha	beta	alpha	beta
Statement	99.57%	99.57%	99.57%	99.54%	91.94%
Branch	85.41%	85.41%	85.41%	86.11%	59.37%

the beta tests do not exercise this code. This is not a concern, as integration tests are not intended to be analyzed in isolation.

6.1 Discussion

Based on the results, the following conclusions can be drawn regarding the established criteria:

- **Independent test execution:** It was possible to achieve isolation without requiring manual effort from the software engineer. When generating the test servers, JS-Distributor automatically detected remote calls and replaced them with local doubles, enabling independent testing. This is particularly important for CI/CD pipelines;
- **Independent endpoint testing:** The newly generated endpoints were successfully tested by the generated test cases that simulate HTTP requests. These tests run locally, and are also suitable for CI/CD pipelines;
- **Integrated test execution:** The monolith's test cases were successfully executed in the distributed version, provided that all servers are running. This demonstrates that JS/Distributor can migrate tests automatically;
- **Ability to detect the same bugs:** The bug detected by the monolith's test cases was also detected by the migrated test cases. Of course, it was a single bug, manually inserted, therefore the empirical evidence is somewhat limited. Ideally, a more robust set of mutants should be used to better compare the bug detection ability of the migrated tests with that of the original tests;
- **Ability to detect new bugs:** The HTTP simulation tests generated by JS-Distributor detected an issue in exception

handling, revealing a bug that was introduced by the migration. This allows the software engineer to fix the bug and prevent faults from occurring. It also revealed a possible improvement to JS-Distributor (automatic encapsulation of exceptions thrown by remote functions); and
- **Preservation of test coverage:** Code coverage remained largely unchanged, indicating that the migration process did not significantly alter the portions of the system being exercised by the tests.

7 Related Tools

In Abgaz et al. [2]'s literature review, the research from Abdullah et al. [1] is cited as one of the few studies that operate directly in the deployment phase. Their approach uses runtime data to identify URLs with similar behavior and group them into microservices. However, rather than physically partitioning the monolithic codebase, their tool clones the entire monolith for each identified microservice and utilizes a load-balancing proxy to redirect requests to the corresponding server. While this method does not fully decouple the source code, a distinct advantage of this proxy-based approach is that it inherently preserves functional equivalence in the new architecture.

Another relevant study by Carvalho and Araújo [4] introduced Node2FaaS, a tool designed to process JavaScript source code and transform local functions into remote AWS Lambda calls. While the authors concluded that the tool aids in migration, they noted that because the user cannot dictate which functions are migrated, some extractions yielded sub-optimal or negative results.

Furthermore, because the migrated function code remains transparent to the callers, the system becomes susceptible to communication/related bugs. Unlike JS-Distributor, Node2FaaS does not provide an automated mechanism to validate the migration. Mono2Rest [6] proposes a two-step migration pipeline. Initially, the tool identifies microservice boundaries by analyzing cohesion metrics, coupling metrics, and semantic similarity, subsequently applying a multi-objective optimization algorithm to this data. The second step utilizes machine learning models to identify the interface of each newly defined cluster. Based on the original function names, the tool automatically generates corresponding REST endpoints by inferring the appropriate URIs and HTTP methods. However, rather than physically decoupling the source code, Mono2Rest merely wraps the existing monolithic logic in REST calls. While this network wrapping could potentially introduce communication-related errors, it can be argued that preserving the original codebase significantly reduces the overall error surface. Similar to other approaches, this tool is confined to Java applications.

Recently, research has explored the use of Large Language Models (LLMs) for code decomposition. For instance, Sellami [8] propose an LLM-based approach that, unlike JS-Distributor, is theoretically language-agnostic, although their implementation and empirical evaluation were confined to Java. However, relying on LLMs introduces a lack of determinism regarding how functions are migrated. To mitigate this, their proposed pipeline includes a step for executing test cases, allowing for LLM-driven or manual corrections before final deployment.

8 Conclusion and Future Work

In this paper, it was introduced a novel feature of JS-Distributor that enhances post-decomposition validation. By combining the generation of simulated HTTP functions with the ability to copy or mock external function calls, the tool enables developers to validate each server independently. It also helps verify whether the newly introduced HTTP layer has not introduced bugs and works as expected. This greatly facilitates the execution of test cases in CI/CD environments. It also maintains the original ability of the test cases to run in the distributed version, allowing integration tests to be conducted. The fact that these tasks are mostly performed automatically, with little to no effort required from the software engineer, is the main contribution of this research.

To evaluate this extension, it was used a modernized JavaScript implementation of the *AcmeAir* benchmark. Following a two-server decomposition scenario, the migrated test suite were executed and demonstrated that it retained the ability to detect a bug introduced in the monolith. It also surfaced an integration error related to HTTP network boundaries that was introduced in the migration process. This highlights the importance of post-migration testing.

The results reinforce the motivation and contributions of this research. There is a known research gap in the implementation and deployment phases of the monolith decomposition process [2]. This includes tools capable of both decomposing a monolith and actively validating the resulting architecture. In practice, JS-Distributor provides a high degree of automation, allowing developers to conduct the migration with less effort and more confidence in the final result's correctness.

The current iteration of JS-Distributor is limited to applications developed in JavaScript. Nonetheless, as JavaScript remains a dominant language in web development, this scope addresses a highly relevant ecosystem. Additionally, the extension did not address AMQP calls. Future work could focus on expanding support to AMQP and including TypeScript.

For future work, we plan to conduct a more robust evaluation, possibly with mutant-based testing, in order to more thoroughly compare the migrated tests' ability to detect bugs with the original ones.

In the next phase, it is planned to extend the tool to automatically generate JavaScript test code designed to collect non-functional metrics, such as performance and scalability. By providing empirical runtime data, this enhancement aims to facilitate and optimize architectural decision-making regarding the newly migrated microservice environment.

Artifact Availability

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

https://doi.org/10.5281/zenodo.21415687

Acknowledgements

This study was financed in part by CAPES (Grant nº 001), and CNPq (Grant nº 406941/2025-4).

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

[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] Luciano Baresi, Martin Garriga, and Alan De Renzis. 2017. Microservices Identification Through Interface Analysis. In *Service-Oriented and Cloud Computing*, Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen (Eds.). Springer International Publishing, Cham, 19–33. doi:10.1007/978-3-319-67262-5_2

[4] 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

[5] Guilherme Escher, Maurício Almeida, João Cardozo, Romeu Leite, Ivan Navas, Vinicius Esperança, and Daniel Lucrédio. 2025. JS-Distributor: decomposing monolith applications into microservices. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 879–885. doi:10.5753/sbes.2025.11016

[6] Matthéo Lecrivain, Hanifa Barry, Dalila Tamzalit, and Houari Sahraoui. 2025. MONO2REST: Identifying and Exposing Microservices: a Reusable RESTification Approach. arXiv:2503.21522 [cs.SE] https://arxiv.org/abs/2503.21522

[7] Ana Martínez Saucedo, Guillermo Rodríguez, Fabio Gomes Rocha, and Rodrigo Pereira dos Santos. 2025. Migration of monolithic systems to microservices: A systematic mapping study. *Information and Software Technology* 177 (2025), 107590. doi:10.1016/j.infsof.2024.107590

[8] Khaled Sellami. 2026. *Automated Migration of Monolithic Systems into Microservices-based Architecture.* Ph. D. Dissertation. Laval University.

[9] Khaled Sellami, Mohamed Aymen Saied, and Ali Ouni. 2022. A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications. In *The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022)*. ACM, 201–210. doi:10.1145/3530019.3530040

[10] Daniele Wolfart, Wesley K. G. Assunção, Ivonei F. da Silva, Diogo C. P. Domingos, Ederson Schmeing, Guilherme L. Donin Villaca, and Diogo do N. Paza. 2021. Modernizing Legacy Systems with Microservices: A Roadmap. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering* (Trondheim, Norway) *(EASE '21)*. Association for Computing Machinery, New York, NY, USA, 149–159. doi:10.1145/3463274.3463334