System Decomposition to optimize functionality distribution in Microservices with Rule Based Approach

Fola-Dami Eyitemi*, Stephan Reiff-Marganiec†
*University of Leicester
fdje2@le.ac.uk
†University of Derby
S.Reiff-Marganiec@derby.ac.uk

***Abstract*—The microservice architecture is an architecture in which a single system is divided into small independently deployed services that are orchestrated together with the use of a lightweight mechanism. Each microservice does not rely much on other microservices (low coupling), but rather on its own resources to perform its task (high cohesion). This paper proposes a novel methodology which decomposes a monolith or other system into microservices in such a way that each microservice will function independently of other microservices while preserving some other key features, with the functionality distribution across each microservice being optimized with regards to usage of the functionality. This methodology makes use of dynamic analysis to identify resources that play a role in enabling the microservice to fulfill its functionality. We establish a set of rules which allows optimized distribution of the functionality. We evaluate the approach by applying it to real systems.**

***Index Terms*—Microservice, Dynamic analysis, Software Architecture**

I. Introduction

The world of software engineering, an ever evolving field with new concepts and technologies, is one in which keeping up with changes is becoming a daunting task. One of its latest arrivals is called Microservices: an architectural design that is best optimized for cloud and edge computing. Fowler and Lewis define the microservice architectural style as 'an approach for developing a single application as a suite of small services, each running in its own process and communicating using lightweight mechanisms, often an HTTP resource API' [8]. These services are built around business capabilities and are independently deployable through fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies [8].

Monolithic applications are applications in which all functionalities required are contained within that one application. These applications, with all their implemented functionality are usually deployed as a single application. While monolithic is an effective architecture from some viewpoints, over time, as an application functionality grows and/or changes, drawbacks became apparent. There was a call for a different architecture that can deal with issues that arose as systems were being scaled; while microservices answered that call it brings up a question of how do you actually split a monolith into microservices or optimize already existing microservices, knowing that the technical implementation does not really help with knowing where to cut. As straightforward as the research questions above seem, there are some drawbacks that will make this challenging as a result of constant change to the monolithic application, they are:

- lack of design and documentation.
- absence of the original design and architectural decisions made during the initial stages of creating the monolith.
- undermining of the original design decisions as many additions and alterations have been made [5].

This work proposes a methodology which provides an in depth view of interactions for every level i.e. packages, methods and classes to help make optimal decisions for decomposition. The resulting decompositions are good microservice systems with high cohesion, low coupling and good sized services with a clear focus on a core functionality. The novel contribution of the paper is a methodology to create an optimised functionality distribution for a microservice system from an existing system. The method is tested using one case study system. Note that due to space requirements the case study analysis is only presented at the package level and class level for a monolithic system to evidence the validity of the approach. The methodology itself is not restricted to decompose monolith systems, but can also be used on already distributed systems to better inform redistribution decisions.

The rest of this paper is structured as follows: Section II gives an overview of related work. Section III gives an insight into the proposed methodology, evidencing why this particular approach is best suited. Section IV will involve the application of the methodology to a system, including an evaluation of the results. Section V concludes the paper and considers some potential limitations and future work.

978-1-7281-6972-9/20/$31.00 ©2020 IEEE
DOI 10.1109/SOSE49046.2020.00015

II. Related Work

Based on [1] we identified an initial set of requirements for a candidate microservices application benchmark to be used – which in turn lead to questions to be taken into consideration when deciding if a monolith is suitable for splitting.

The questions that were identified are:

- does the monolith have the propensity to be divided into multiple interfaces in which each interface are connected to each other?
- if there is high level of coupling present in the monolith?
- can multiple aspects of business logic with some distinct implemented functionalities be identified in the monolith?
- is there a big possibility for growth and modifications to be made to the system?
- is there a benefit to distribute the application through a cloud/ fog system?

Migration from a monolithic application to microservices architecture has been frequently looked into as a research area and some solutions and methodologies have been created; we will now review the most relevant ones.

[7] emphasizes the choice of granularity, which is the breakdown of the microservices obtained from a monolith as the main driver to balance between the costs of quality assurance and the cost of deployment; this is important as there is no definite size for a microservice. If the coupling between the functionalities in a monolithic service are massive a huge cost on quality assurance is incurred as making a change to a function will have major impacts on other functionalities; however when breaking into microservices there is an increase in the cost of deployment as each service will be deployed individually yet a lower cost of quality assurance can be achieved as there is little or no modification to the other services when modifying a particular service.

[9] created an extraction model in which there are three stages which are the monolith stage, the graph stage and the microservices stage. A step called the construction step transforms the monolith into the graph representation and a clustering step decomposes the graph representation of the monolith into microservices candidate. The construction step occurs by gathering some informations like set of class files, history of changes made and the set of developers that contributed code to the monolith. This information is then used to plot a graph. The graph goes through the clustering step using some extraction strategies; these extraction strategies are a Logical Coupling Strategy in which class files that are changed together during a particular modification are put together, a Semantic Coupling Strategy in which classes that contain code about the same domain model entities like variable names and method names are put together in the same microservice and Contributor Coupling Strategy which analyzes the authors of the changes on the class files in the version control history of the monolith and using this breaks the monolith into microservices;

[4] presents an extensive work on feature location and feature comprehension which deals with identifying where exactly the execution of a particular feature starts and ends using a visual representation of traces. It also provides an understanding of the interactions that take place during a feature invocation.

[3] also looks into decomposing systems into microservices, but focuses on decomposing monoliths by breaking down business logics into microservices candidates by making sure any resulting microservice has only one output data which can then be transferred to the next microservice for further work. The issue with this is it takes a look at this from a high level approach without actually taking a look at the system. It is an ideal technique when looking at system that are relatively small and focused on specific outputs.

III. Methodology

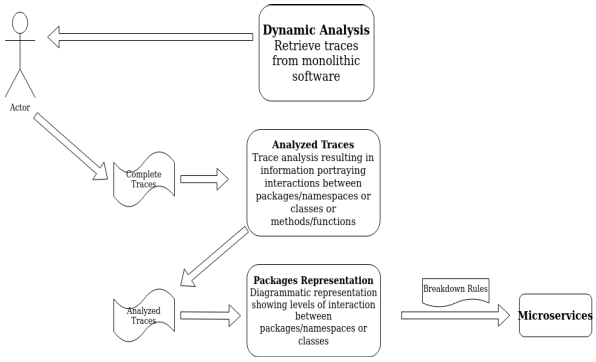


Fig. 1: Process Model

When decomposition is done from any other architecture into the microservice architecture it is important to note that an important basis for decomposition will be how independent is each microservice after breakdown. When we talk about independence, we mean what is the level of coupling between different microservices, what is the level of cohesion within each individual microservice. The question then is which technique of decomposition will result into a well optimized microservice. This section will look into the proposed methodology for dealing with this question.

A. Dynamic Analysis Extraction process

Dynamic analysis is the analysis of data gathered from a running program. This has the potential to provide an accurate picture of a software system, amongst other benefits, because it can reveal object identities and occurrences of late binding [4] and really highlights which components are closely related in terms of run-time behaviour rather than their design. In this process the aim is to exhaust every possible business capability while extracting the traces. The importance of dynamic analysis being used in this context is quite significant as it will give the opportunity to view interactions which would not otherwise be seen in the design due to various reasons. This process is dependent on all the functionality of a system being implemented during the trace extraction process. It is advisable to gather traces over a long period of time. There

are sufficient tools and libraries that will help log traces based on the language or framework used.

B. Representation techniques for Extracted traces for package level (Top Level)

Providing a high level view of interaction between different packages is very important. A package is an organized and functionality based set of related interfaces and classes [10]. The traces are put through a simple software tool which then comes out with a more streamlined set of traces which focuses on the package making a call and the package being called for a line of interaction. This streamlined set of traces is then processed by a software tool which outputs the following

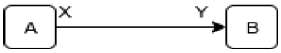


Fig. 2

In Figure 2 above,

$$X = \frac{Callsmade from package A to package B}{Callsmade by package A} * 100$$

x is the percentage of calls made from package A to package B over the number of calls made by package A in total

$$Y = \frac{Callsmade to package B from package A}{Callsmade to package B} * 100$$

and Y is the percentage of calls made to Package B from package A over the total number of calls made to package B

This provides the high level representation of interaction needed for the next phase of the method.

C. 6 Rule approach for the package level (Top Level)

With the high level interaction represented, the next phase is to retrieve the microservices from the output of the previous step. For this step a total of 6 rules were identified to help in making decisions on where to break into microservices. The percentage values used in the rules definition were acquired as a result of analysis into different systems and those set of numbers have proven to provide consistently good results without contradicting the effectiveness of the rules. These rules are based solely on the level of interaction and where cutting will lead to an optimal set of microservices being created. These rules are

1) First Rule: If the percentage of calls being made by A to B is around 70% and above and the percentage of B being called from A is also around 70% and above where A and B are packages.

Since both A and B are highly dependent on each other, they should be brought together to make up or populate a microservice.

If percentage of calls made to package B from package A are the only calls to package B (regardless of the value of x)

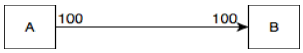


Fig. 3: Rule 1

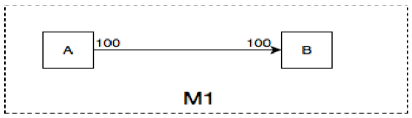


Fig. 4: Rule 1 Breakdown

meaning that no other package calls B and additionally B does not make calls to any other package, then package B should be with A.

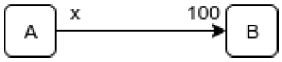


Fig. 5

2) Second Rule: If the percentage of calls from A and C to B are low to high (i.e. between 1 and 100 percent) and B is being called by both A and C only with the percentage of either A or C being 50% or higher.

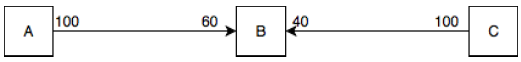


Fig. 6: Rule 2 Breakdown

Then B can be shared into B^{I} and B^{II} where B^{I} holds the data specifically needed by A and B^{II} holds the data specifically needed by C

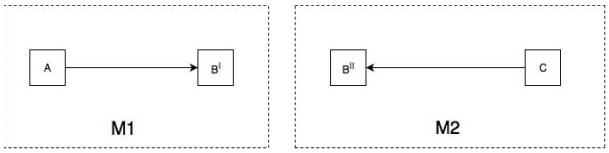


Fig. 7: Rule 2 Breakdown

3) Third Rule: If the percentage of calls from A,C,D to B are low to high (i.e. between 1 and 100 percent) and then all the percentages of B being called by A,C and D individually are less than 50%.

Then B can be duplicated across A, C and D to each create a microservice.

4) Fourth Rule: If the percentage of calls from A and C to B are low to high (i.e. between 1 and 100 percent), and B

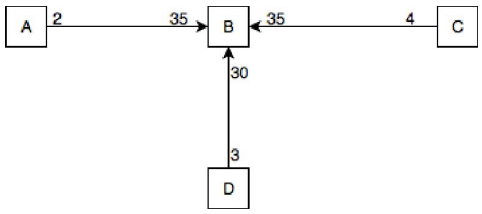


Fig. 8: Rule 3

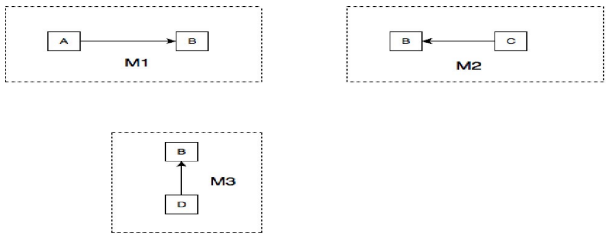


Fig. 9: Rule 3 Breakdown

is being called from A and C, and percentage of calls to D being made by B is also 100%.

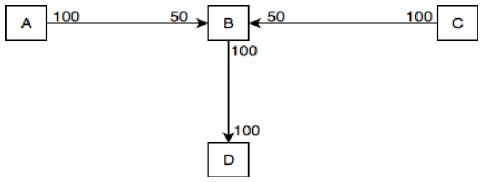


Fig. 10: Rule 4

Then both B and D can be duplicated and added to both A and C to create microservices.

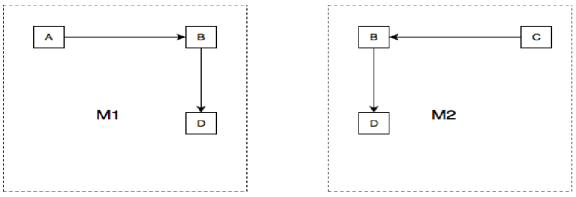


Fig. 11: Rule 4 Breakdown

5) Fifth Rule: If the percentage of calls from A to B and percentage of B being called by A are not too significant but both A and B make various calls to other packages and also gets called by other packages. This rule is still valid for use if A and B are not called by any package.

A and B should be separated.

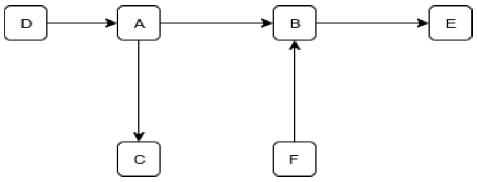


Fig. 12: Rule 5

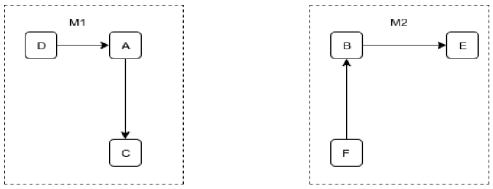


Fig. 13: Rule 5 Breakdown

6) Sixth Rule: If the interaction between two packages is one in which they both have the interaction going both ways to and from each other.

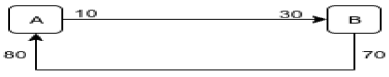


Fig. 14: Rule 6

it is important then to take into consideration the state of the surrounding packages interaction in making decisions on where to cut.

For example you could take into consideration

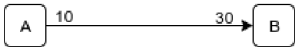


Fig. 15: Rule 6 Breakdown

and making use of the fifth rule separate package A and package B if they each are already connected to other packages to make another microservice knowing that if you leave them together, it will make that microservice bigger in comparison to the others.

You can make use of the first rule and keep package A and package B together, especially if package A have few or as little as one other package it calls.

Fig. 16

D. The Class breakdown criteria

When looking into the class breakdown rule, one criteria was identified to keep the breakdown simple and not have to deal with complex breakdown issues. The criteria is it is advisable to breakdown packages that have no call outside of itself but rather only calls itself or is called by other packages. Taking a look at the example used in expressing the second rule in the 6 Rule approach, the Monolith is

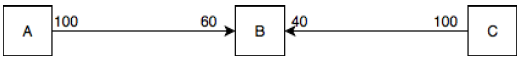
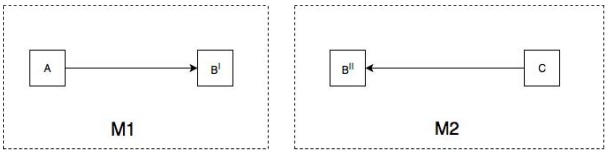


Fig. 17

while the resulting microservice is



Package B never makes any outside call but rather it is being called by other packages, which makes it a perfect candidate for class breakdown. The resulting microservice showcases that the classes within the package B have been broken down to better accommodate what package A and package C needs, since the package B is altered for both package A and package C the new packages formed are B^{I} and B^{II} consecutively.

To extract B^{I} and B^{II} from the traces, the first important thing to do is isolate every interaction between package A and package B and likewise package C and package B and in cases where package B calls itself also isolate every interaction package B has with itself, this set of isolated traces together then becomes the set of classes you use to form package B^{I} and package B^{II}.

E. 6-Rule Approach Application

When applying these rules, a logical question that arises is in which order should these rules be applied during the decomposition process. During decomposition, we follow these steps:

1) First Step: Identify packages that fit the fifth rule, which are packages that are called by other packages and also makes calls to other packages. Applying the fifth rule identifies the packages that will serve as base packages for each microservice.

2) Second Step: Now with the identified packages, the next step is to look at the connections to and from each identified package. Making use of this connections, identify which rule it fits with and apply individually. This step can be carried out differently but the results are identical.

F. Explanation of representation techniques for Extracted traces for class level

For the purpose of this paper, the extracted traces are represented at the class level. The model that is used for the representation is

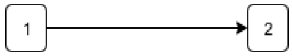


Fig. 18

where 1 and 2 represents the class and the arrow represents the direction of interaction as the class represented by 1 calls the class represented by 2.

IV. Case Study and Evaluation

This chapter will look into two systems and the application of the methodology explained in the previous chapter on them.

A. eShoponWeb

eShoponWeb is a web based ASP.NET core shopping cart software. This version of the software that will be analyzed was created with .NET Core 2.2 framework [2].

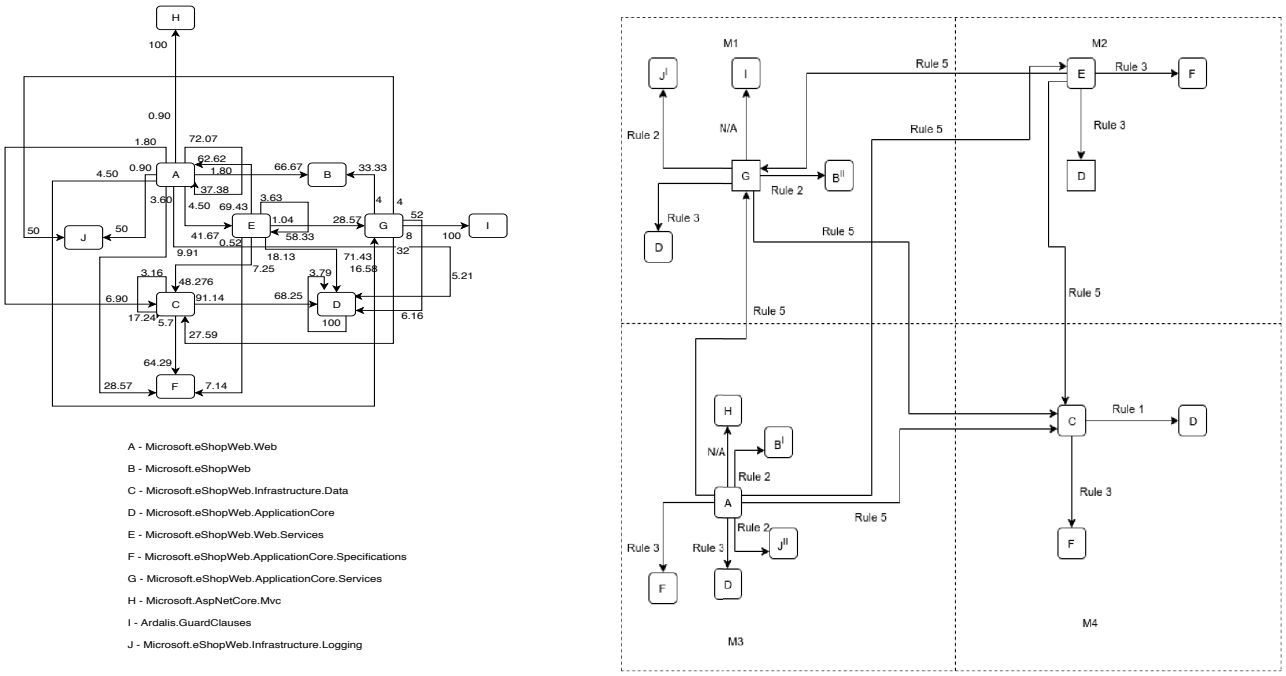
1) Dynamic Analysis Extraction Process: PostSharp is a pattern-aware compiler extension for C# and VB, this was used to extract the traces. The traces were acquired over a period of time but the completeness of the extracted traces cannot be guaranteed as there might be some parts of the software that are not reached as their pre-conditions might not have been met, this is the major drawback of making use of Dynamic Analysis. [6] has a link to the extracted traces.

2) Representation of extracted traces: The traces are parsed through a software tool which then isolates each package and then calculates X and Y for each connection between each package, making use of the formulae given in Part B of the methodology chapter. [6] also has a link to the resulting traces

With the resulting trace, the representation is formed. Fig 19a is the resulting representation.

3) 6 Rule Approach Application: With the representation in Fig. 1a, we can now apply the rules identified in section C of the Methodology chapter. In section E of the methodology chapter, the first step is to identify the packages that fit the fifth rule, which are packages that are called by other packages and may also make calls themselves to other packages. This packages are identified as A,C,E and G.

The next step is to look at connections to and from A,C,E and G; making use of 6 rules approach, identify where each fits in and make the required change. The final decomposed set of microservices can be seen in Fig. 1b. We can notice the connection between package G and package I says N/A, this is because I is an external library.



(a) eShoponWeb Monolith Representation

(b) eShoponWeb Microservices Breakdown

Fig. 19: eShoponWeb Packages Breakdown

4) Class level Decomposition: To decompose in class level, as identified in section D of the methodology; it is advisable to breakdown packages that has no call outside of itself but rather only calls itself or is called by other packages. The two packages that fits this criteria in fig. 1a are B and J.

Next step is to pick one of the package, for example J. In the decomposed microservice fig. 1b, B was broken down into B^{I} and B^{II} where B^{I} holds the data needed by A and B^{II} holds the data needed by G.

To extract B^{I} from B, a simple tool was written to go back into the traces and extract every communication between B and A and also between B and itself. A represents the microsoft.eShopWeb.Web package and B represents the Microsoft.eShopWeb package. The resulting data being represented by the model defined in section F of the Methodology chapter can be seen below.

The above image shows the classes within Package A and package B that are connected, this means B^{I} will only consist of the Microsoft.eShopWeb.CatalogSettings.

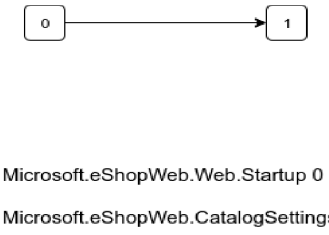


Fig. 20

V. Discussion

Making use of this approach on the systems above has proven to be effective as the approach ends up generating a breakdown, bringing classes together that are highly cohesive to make up a microservice, this goes a long way in achieving the goal of having truly independent microservices when implementing various functionalities, which is the research question looked into in this paper.

The six rules identified so far appear to be exhaustive but there is still the chance that while decomposing other systems there might be scenarios encountered in which none of the rules will fully cater to the situation, which suggests the possibility for growth with this particular approach. Decomposition on class level is also shown to be effective as only packages which are not closely coupled with other packages are decomposed leaving a small margin for error.

This approach is quite straightforward with its application and easy to implement, a large skill set will not be required to implement this approach. It can also be applied to any system regardless of the tools used for its creation as all that's needed is the extracted traces.

The resulting microservice decomposition for the both systems show an architecture in which each microservice is self sufficient, i.e. (the resources it needs to function are available within that service) and hence the resulting system will function optimally. Each microservice in the resulting architecture is also small and autonomous as each microservice will be able to do one thing well to fulfill a business functionality, while each microservice implements a relatively independent piece of business logic making it highly cohesive [3].

It was identified during the analysis of the system, that the traces gotten might not fully showcase every interaction possible within the systems, this is potentially its major drawback which could be looked into in the future.

VI. Conclusion And Future Work

The defined methodology provides a structured way to analyse the runtime behaviour of a system and represent the coupling between parts at various levels, showing the feasibility of migration to microservices, the detailed dependency between classes (the box diagrams) and the resulting microservices oriented functionality distribution. The methodology could then be augmented with a final step of system refactoring that follows the schema from the distribution step to re-cluster the code accordingly this techniques were not included, as refactoring are fairly standard nowadays.

A challenge with this methodology is for an accurate view of the base system to be retrieved before decomposition hence there is a need for a detailed dynamic analysis that covers every aspect of the system. For the best effectiveness, it would be ideal to retrieve the traces over a period of time while the system is deployed and actually being used by consumers. A problem identified for a dynamic analysis [4] is the large amount of trace data collected and the need to analyze this. However, the analysis required in our work is quite straight forward and supported by tools that produce easily accessible summary information.

The rules identified and shown in this paper are the result of detailed analysis of a number of systems; they have shown to be sufficient and complete. For the future work, we will be looking into static analysis as a technique to support dynamic analysis for the traces extraction process for a more exhaustive trace of the system. An Empirical study could be done to a lot more systems than that covered in the scope of this paper to further validate the percentage numbers given when defining the rules.

References

[1] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi. Benchmark requirements for microservices architecture research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13, May 2017.

[2] S. ardalis Smith. eshoponweb as open-source project. https://github.com/dotnet-architecture/eShopOnWeb. Accessed: 2019-11-19.

[3] R. Chen, S. Li, and Z. Li. From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, Dec 2017.

[4] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 49–58, June 2007.

[5] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas. Towards the understanding and evolution of monolithic applications as microservices. In *2016 XLII Latin American Computing Conference (CLEI)*, pages 1–11, Oct 2016.

[6] F.-D. Eyitemi. Breakdown console. https://github.com/lorddamsy1000/BreakdownConsole, 2020.

[7] J. Gouigoux and D. Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 62–65, April 2017.

[8] J. Lewis and M. Fowler. Microservices a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. Accessed: 2018-01-22.

[9] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531, June 2017.

[10] T. Website. What does package mean. https://www.techopedia.com/definition/24011/package-java. Accessed: 2019-11-20.