Migration Process from Monolithic to Micro Frontend Architecture in Mobile Applications

Quentin Capdepon[1,2,*], Nicolas Hlad[1], Abdelhak-djamel Seriai[2] and Moustapha Derras[1]

[1]Berger-Levrault, Toulouse, France
[2]LIRMM, University of Montpellier, France

Abstract

Today mobile applications are often monolithic architecture that are complex to maintain, especially when they reach an industrial scale. Micro FrontEnd (MFE) architecture offers an opportunity to re-architecture systems into smaller units. However this re-architecturing is often manual and as yet to be adapted on mobile application developpement. This paper introduces early ideas and plan to migrate monolithic mobile architecture to MFEs using the model-driven engineering (MDE). Our approach is tailor to work for mobile Flutter application and uses a Dart meta-model based on Famix. Alongside our process description, we expose the MFE identification challenges, limitations, and future work needed to achieve it. If implemented, this migration would have the potential to leverage MFE's advantages to mobile applications, leading to improvement in the development practices at an industrial scale.

Keywords

Micro frontends Architecture, Architecture migration, Model Driven Engineering, Moose Famix

1. Introduction

Mobile applications are an essential part of our daily lives, with 225 billion downloads globally in 2022 [1]. However, industrial mobile applications become more complex. For instance, our industrial partner Carl Software/Berger-Levrault commercializes a GMAO Flutter mobile application of over 400k lines of code. Its developers reported maintainability and scalability issues, which they attributed to the monolithic architecture of their application.

Flutter is a multi-platform SDK developed by Google [1]. It allows the creation of natively compiled mobile, web, and desktop apps from a single code base developed in Dart [2]. Unfortunately, Flutter applications tend to have a monolithic architecture, a design notorious for its complexity in being maintained and evolved over time [2].

In our work, we look at ways to modularize the initial monolithic architecture by utilizing

IWST 2023: International Workshop on Smalltalk Technologies. Lyon, France; August 29th-31st, 2023
*Corresponding author.
qcapdepon@lirmm.fr (Q. Capdepon); nicolas.hlad@berger-levrault.com (N. Hlad); Abdelhak.Seriai@lirmm.fr (A. Seriai); mustapha.derras@berger-levrault.com (M. Derras)
https://www.lirmm.fr/~seriai (A. Seriai); https://www.research-bl.com (M. Derras)
0000-0003-4989-2508 (N. Hlad); 0000-0003-1961-1410 (A. Seriai)

[1]https://www.statista.com/topics/1002/ mobile-app-usage/
[2]https://web.archive.org/web/20230504145028/https://en.wikipedia.org/wiki/Dart_(programming_language)

micro frontend (MFE) architecture. This architecture is particularly suited to our needs : i) MFE modularizes frontend applications, which corresponds to the nature of mobile applications [3]; ii) MFE are loose modules, which contributes to their development autonomy, thus facilitating their maintainability [4]; iii) MFE can be composed to form a global application and have a reusability potential in different apps. However, using MFE presents two problems : a) Today, MFE architecture is mainly used in the industry for web applications; b) In the domain of mobile applications, the scientific literature lacks significant contributions addressing the migration from monolithic architecture to MFE architecture.

As part of our initial work, we developed an MFE shell for mobile called MicroFrontendShell with our industrial partner Berger-Levraut. The shell is a key component of the MFE architecture. It is a container application that provides the framework for loading and managing multiple MFEs within one front-end application. The Shell manages a consistent user experience across all micro frontends, managing routing and page transitions, and facilitating communication between different micro frontends [3]. Thus, our goal is to design a migration process that will target an MFE architecture compatible with the shell.

Our migration approach is a work-in-progress concept and this paper only presents a plan to achieve it using existing Pharo tools. We focus on the identification steps, the transformation and generating Dart source code from the obtained model is out of the scope of this paper. In this early work contribution, we propose : i) a general description of a migration process based on a model-driven engineering (MDE) approach using Moose[4]; ii) implementation details regarding the technologies best suited for the MFE identification step; and iii) a discussion on the challenges regarding the feasibility of our migration process and the feasibility of MFE on mobile in general.

Our paper is structured as follows: section 2 provides a background on the monolith and microarchitectures concepts; section 3 gives an overview of our migration approach and its steps; section 4 discusses the identification steps and its challenges; Finally, section 5 examines the limitations of our process, and section 6 presents the related works to our paper.

2. Background

In this section, we provide background on micro frontend and monolithic architectures.

Monolithic Architecture. Monolithic architecture has historically been present in most programming languages and applications [5]. The entire application presenting itself as a single, cohesive unit. While its easy implementation is advantageous in the early stages of a software application's life, it can pose significant challenges in evolution and maintenance for larger applications. The presence of legacy code from years of iterative development makes it difficult to maintain and evolve the monolith. Thus, monolithic architecture is often identified as the cause of slower development cycles, a higher risk of errors, and difficulties in scaling the application [2, 5].

[3] https://web.dev/learn/pwa/architecture/
[4] https://modularmoose.org/research

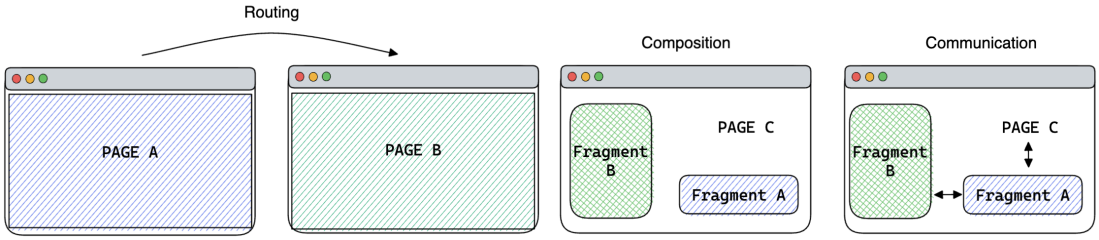

Figure 1: Core concept of micro frontend by M.Geers [3]

Micro frontend. Micro frontend architectures are a way of building front-end applications as a collection of loosely coupled, standalone modules, each called micro frontend. Each MFE is a discrete feature or business capability of an application that is designed, tested, implemented, and maintained independently of other modules. This architectural approach is recognized for improving the agility as well as the scalability of front-end development, increasing team autonomy by allowing each MFE to be owned and managed by a separate team [4]. Micro frontend being directly linked to the frontend, they are easier to consider as *Pages* (i.e. an entire Single Page Application) or *Fragments* (*i.e.* a widget that composed a page) [3]. MFEs are not limited to the UI page or UI fragment: their perimeter includes all the stacks that start from the UI to the database. This allows each team responsible for the MFE to fully own their stack, thus being independent.

To integrate the micro frontends into a unified user interface, a MFE architecture typically uses three core concepts: *routing*, *composition*, and *communication*. These concepts are illustrated in Figure 1. Routing refers to the mechanism by which the user navigates between micro frontends. Each MFE typically has its routing configuration, dynamically or statically loaded depending on application requirements. Composition refers to the act of combining multiple MFE to form a UI page. On one hand, we talk about *server-side composition* when the composition is made on the server before it is sent to the client. On the other hand, a *client-side composition* is when a shell is required to collect and assemble all the MFEs by executing a script on the client side. Finally, communication refers to how MFEs exchange data and events with each other. This communication can be direct, with MFEs communicating with each other without any intermediary, or indirect, with MFEs going through a remote backend or a local shell to exchange their messages [3].

3. Migration Process Overview

Our migration process is illustrated in Figure 2. It is composed of the following steps.

Step 1: Analysis. The first step takes the monolithic Dart source code as input and outputs an instantiated DartFamix model from it. Pre-requirements for this step are the creation of a Dart parser, the definition of a Dart metamodel (based on Famix), and the implementation of a *Famix Dart* importer (to instantiate the Famix Dart metamodel). Figure 2 illustrates this step by analyzing the dart code of a class *C1* and turning it into a Famix Dart model. This model captures the dependencies of the entities following the Famix Meta model [6].

Step 2: Identification. The identification step requires as input the Famix Dart model instantiated from the monolith and outputs the clustering of the model's entities to create a MFE architecture. Visualizations are essential in our migration process for informing developers about the impact of changes on the architecture. This step's pre-requirements are the creation of visualizations, the definition of clustering metrics, and the ability to label model entities according to the clustering. The identification process includes the detection of dependencies, points of interest for grouping, the detection of possible clustering, the creation of visualizations, and the computation of cohesion metrics [7], all of them are hinted at in section 4. It also detects inter-group dependencies which are seen as *violations* and are resolved by the transformation step. As in Figure 2, the identification clusters the model entities here based on their proximities with class *C1* and *C2*. Thus, in this example, creating two micro frontend candidates: *MFE1* and *MFE2*. Here a violation appears from the invocation that remains from M1 to M2. Re-architecturing the monolith imply to transforming this invocation inside the new MFE architecture.

Step 3: Transformation. The third step, transformation, takes as input the clusters of entities from the identification and outputs a refactored model for the MFE which resolves the violations. This step requires the definition of rules for inter-grouping violations and their resolution. The transformation involves reordering the model entities according to each cluster. Our interest is to find automated ways to resolve most of the violations. However, it is likely that some will involve the developer's input [8]. In our example, we see that there is still a dependency between *MFE1* and *MFE2*. The transformation goal is to resolve this violation by, for instance, redirecting the dependencies through our MicroFrontendShell. Once resolved, a code source exporter will generate the MFEs code, ready to be deployed.

The migration from a monolithic architecture to micro frontends introduces various challenges that must be addressed to ensure a successful transition. Since this paper focuses on early work, it only covers the identification step and leaves the transformation for future works

4. Identification

This section discusses ideas and challenges related to the identification phases within our migration process. This step constitutes the core of our future work and involves the scientific challenges we strive to address. We also assume that we have a tool that parses Dart code and produces a Famix model for Dart, as shown in section 3.

To migrate monolithic Dart code to MFE, we need to tackle the challenges depicted in Figure 2, which are part of the identification step: i) Creating visualizers of our Dart model; ii) Clustering the model entities to identify potential MFE.

The identification step is a semi-automatic process involving automated techniques and expert validation. The main objective is to propose a rearrangement of the entities within the model, aiming to create clusters of entities to modularize the initial architecture into an MFE architecture. The validation based on the expert's approval is essential since architectural migration is a risky process [9]. Thus, supports like visualizations help experts in their decision-making.

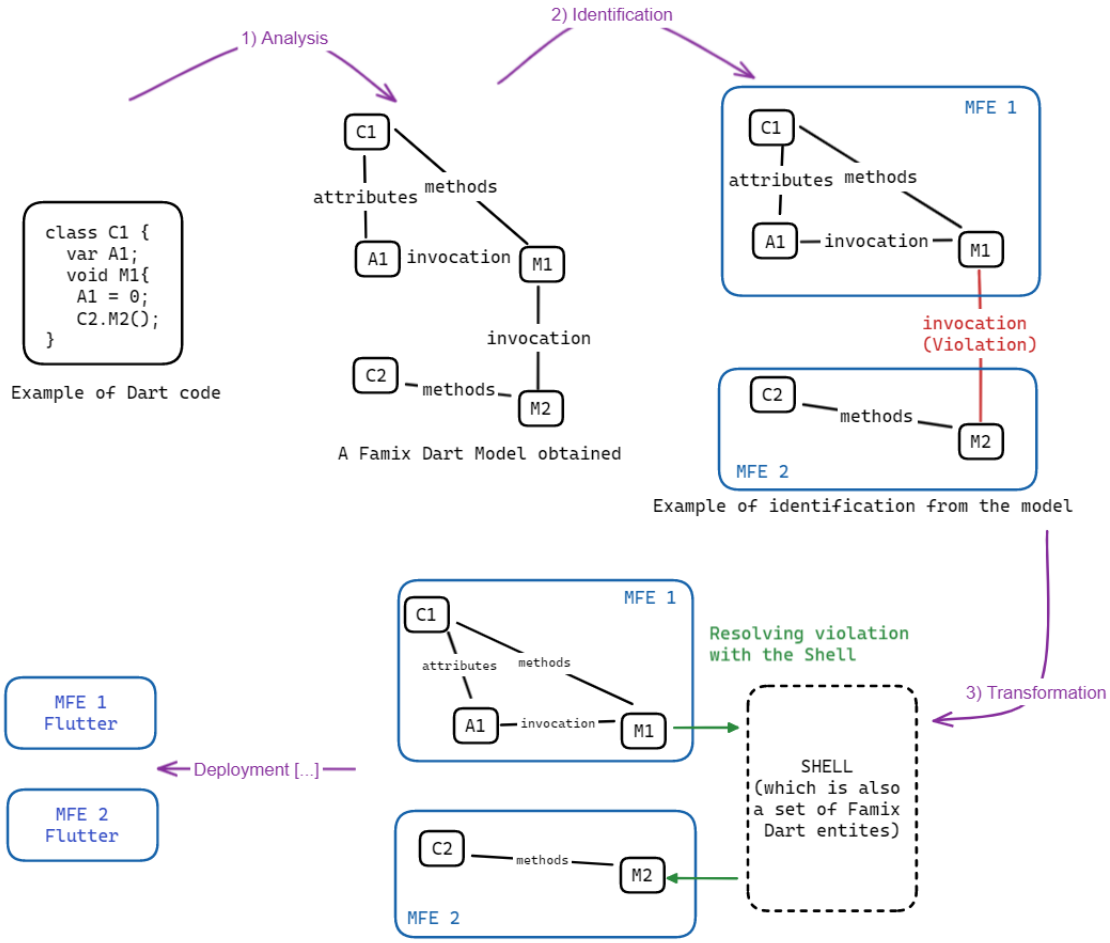

Figure 2: Migration overview illustrated on a basic example

4.1. Clustering

With this step, we aim to propose a clustering approach that maximizes key principles of MFE architecture. We translate these principles into three criteria, defined as followed :

The *Cohesion*, ensures that each MFE encapsulates its functionality, user interface, and assets. Within an MFE architecture, features represent self-contained units of functionality. We aim to use the import declaration within a Dart file to determine which files are required to run a particular feature, thus forming an initial grouping of MFEs candidates. Expert validation is required to evaluate the feasibility and the cohesiveness of the MFEs candidates.

The *Modularity*, implies the creation of loosely coupled modules through clustering. To get modular MFEs candidates, we need to visualize the violation between all the different MFEs candidates and resolve them, which may involve decoupling business code, duplicating existing code, establishing communication protocols for data sharing, and generating code. By employing this technique, we want to address the MFE principle of independent development

and deployment of individual front-end modules.

The *Reusability*, is centered around maximizing the reusability of the candidate MFE. This principle emphasizes that an MFE should be self-contained and capable of independent use and development. The objective is to develop MFEs that can be utilized both individually and in collaboration with other MFEs across various projects, including those originating from different monolithic architectures. To achieve this, we want to analyze the Dart code to identify fragments that indicate the presence of user screens, mains, and data sharing. This fragment will help us to build a complete and feasible MFEs cluster. Expert validation is necessary to evaluate the potential for reusing the proposed clusters and ensuring their autonomy and compatibility for integration into various contexts.

This joint effort enables us to improve and refine our yet-to-be-determined metrics. However, it is important to note that interaction and adjustments made by experts (with the help of our visualization) can introduce new violations.

4.2. Roassal visualizer

Our first challenge concerns the creation of interactive visualization that helps expert validation of the MFE architecture during the identification. We need to investigate diverse approaches for representing the structure and dependencies within the Dart model. To do so, we must be able to represent the monolith from different points of view to allow them to make the best decisions about the MFEs to be created. Among these views, our work will focus on views that expose the fundamental principles of MFE (routing, communication, composition). We can mention a view highlighting the navigation between the monolith's screens and a second highlighting the relationship between the monolith's code and the developers' work. Here we present two possible visualizations for the identification: one focuses on the frontend navigation; and one on the git contribution. Potential visualizations for the identification process could include navigation and a git contribution representation.

4.2.1. Navigation Graph.

This visualization represents the navigation flow within an application. This visualization can show how different screens or components are accessed and navigated, providing insights into the overall structure of the application and potential boundaries for MFE separation. Though, designing a navigation flow graph for a Dart application in Flutter poses challenges due to the widget composition employed in Flutter. Getting information about widget types is essential, but it is only available in a *widget tree*, which is constructed at compilation time. Thus, a dynamic analysis of the runtime application is required to extract the widget tree. Here, the challenge is to combine the Dart model obtain by static analysis, with the navigation model obtained by dynamic analysis. It requires efficiently merging the information of these two analyses to capture how the model entities are grouped and accessed during navigation.

The concept of this visualization is represented in Figure 3. Inside we see a basic music player application concept with three screens: a list of albums, a list of songs within an album, and a media player. The *lateral navigation* indicates that the application provides a way to navigate freely between the green and the blue screens (e.g. using a bottom navigation bar). While the

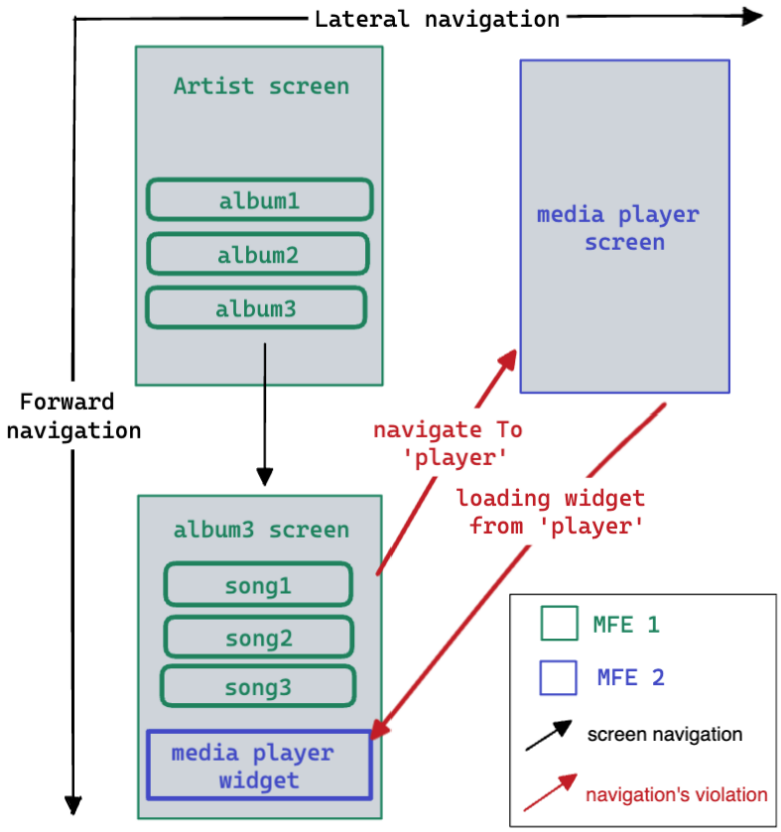

Figure 3: A concept for the *Navigation visualization*

forward navigation indicates that *album screen* is only accessible after selecting an album inside *artist screen*. In this clustering example, *Album and Artist* screen is associated with MFE1, while the *media player* is associated with MFE2. We see the navigation represented as an arrow. In this concept, we represent the navigation between the two MFEs as a violation (in red) since it creates direct dependencies between the two MFEs. Thus, experts can discuss the initial clustering and decide if *media player* should belong within *artist* and *album screen* MFE.

4.2.2. Git Contribution Analysis

This visualization can leverage data from version control systems like Git to analyze developer contributions to specific files. This visualization provides insights into which developers have worked on different parts of the codebase. Additionally, it helps identify potential boundaries for the MFE division. Separating the team that works on different parts of the monolith becomes essential in achieving one of the key principles of MFE architecture. To successfully implement an MFE architecture, each MFE will require a dedicated team of developers who have previously worked on the corresponding codebase. This ensures that the team possesses the necessary expertise and knowledge to effectively maintain and enhance their respective MFE.

This promotes independent development and the overall success of the architectural approach.

Our future works focus on implementing these two visualizations using Roassal. Roassal allows the creation of interactive visualizations within Pharo and links them to a model. Therefore, we can propose expert interaction and translate these interactions to our model using Roassal. As for any work on software visualization, the challenge is to fine-tune our visualization so that it offers meaningful information for the experts. Thus, we need to conduct an evaluation of our work with real-life developers and industrial projects, which is often the most challenging part of software engineering research.

5. Limitations

Our migration approach has limitations that we anticipate and will need to address. These limitations can be classified into three categories related to different stages of the migration process.

Firstly, there is a risk of frequent changes in the Dart grammar and the Flutter framework, which can make some of the features on which the identification is based obsolete [5][6]. The creation and maintainability of a parser and meta-model for Dart is in itself a challenge considering the rapid evolution of this language. Let alone that Flutter is also evolving. This limitation could be addressed by regularly updating the identification tool to ensure compatibility with the latest versions of the Dart language and Flutter framework.

Secondly, the non-autonomous deployment of MFE is a significant limitation of the approach. All MFEs are natively integrated into the deployed app, which contradicts the "autonomous deployment" aspect of a micro frontend. However, this constraint is intrinsic to the native mobile platforms of today (iOS and Android) and cannot be bypassed. A potential solution path could be to use a platform such as Shorebird[7] that seems to allow Dart code injection within an already deployed app.

Thirdly, when performing a static analysis only on the Dart part, we are not able to analyze Flutter platform-specific code and configuration files. The platform-specific code, written in languages like Kotlin, Java, Objective-C, or Swift, and the configuration file (pubspec.yaml) provide important insights into the application's behavior and dependencies. Analyzing only the Dart code would provide a partial understanding of the codebase, potentially missing issues or behaviors specific to the platform or configuration. Therefore, the restricted analysis scope of Flutter's static analysis tools limits the comprehensive assessment of the application's overall code quality and behavior.

6. Related Works

Peltonen and al.[4] conduct a literature review on the adoption of MFE in the industry. They show that most teams adopt MFE to reduce the maintenance complexity of their frontend application. In his book, Geers offers an extended definition of MFE based on his extensive

[5]https://docs.flutter.dev/release/breaking-changes
[6]https://docs.flutter.dev/release/archive
[7]https://github.com/shorebirdtech/shorebird

industrial experience [3]. We took inspiration from his work and we work on adapting its definition for mobile using Flutter. In papers and keynotes, Mezzalira et al. share their experience on anti-patterns in MFE [10, 11]. We plan to study these anti-patterns since they may help us identify violation rules during our migration process. Unfortunately, we have yet to find studies or reports that cover MFE on mobile.

We found two industrial frameworks for MFE on mobile. The first one by Braz, is an open-source Flutter package to organize a Flutter project as a set of MFE, called micro apps [12]. The second is an extension of the MFE framework of *ionics* [13] where MFEs are embedded for mobile applications. However, both frameworks only focus on mobile MFEs built from scratch, whereas our goal is to build them from our migration process. In future works, we will look at the implementation of Braz and study how our micro app deployment target differs from his proposal.

Finally, our migration process takes inspiration from works on migrating monolith to microservice [14, 15, 16, 17, 18]. Noticeably, we adapt to mobile MFE the work of Zaragoza in [8], who presents a similar migration approach for microservices from the identification to the deployment.

7. Conclusion

Our article presents a novel approach for migrating monolithic mobile Flutter applications to micro apps using MDE. The migration consists of three steps: analysis, identification, and transformation. We describe several challenges, such as developing new visualization and clustering. Our goal is to develop this migration approach for the next three years, with a release of the Dart Famix meta-model by the end of 2023. After that, we plan to focus on clustering methods and study anti-patterns in MFE to detect violation rules during the migration process. Once we reach the deployment step, we intend to study the impact of micro frontends on mobile app development. We especially, want to investigate the impact of MFE architecture on performance, user experience, and developer productivity. With this research, we aim to provide a practical and effective solution for migrating monolithic mobile applications to micro apps using MDE.

References

[1] Flutter, Flutter releases, 2018. URL: https://docs.flutter.dev/release/archive.

[2] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: Yesterday, Today, and Tomorrow, Springer, 2017, p. 195–216. URL: https://doi.org/10.1007/978-3-319-67425-4_12. doi:10.1007/978-3-319-67425-4_12.

[3] M. Geers, Micro Frontends in Action, manning publications ed., 2020. URL: https://livebook.manning.com/book/micro-frontends-in-action/.

[4] S. Peltonen, L. Mezzalira, D. Taibi, Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review, Information and Software Technology 136 (2021) 106571. doi:10.1016/j.infsof.2021.106571.

[5] R. Stephens, Beginning Software Engineering, 1st ed., Wrox Press Ltd., GBR, 2015.

[6] S. Tichelaar, S. Ducasse, S. Demeyer, Famix and xmi, in: Proceedings Seventh Working Conference on Reverse Engineering, 2000, p. 296–298. doi:10.1109/WCRE.2000.891485.

[7] J. Al Dallal, L. C. Briand, A precise method-method interaction-based cohesion metric for object-oriented classes, ACM Transactions on Software Engineering and Methodology (TOSEM) 21 (2012) 1–34.

[8] P. Zaragoza, A.-D. Seriai, A. Seriai, H.-L. Bouziane, A. Shatnawi, M. Derras, Refactoring monolithic object-oriented source code to materialize microservice-oriented architecture, in: ICSOFT, 2021, p. 78–89. doi:10.5220/0010557800780089.

[9] A. Selmadji, A. Seriai, H. Bouziane, R. O. Mahamane, P. Zaragoza, C. Dony, From monolithic architecture style to microservice one based on a semi-automatic approach, in: 2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020, IEEE, 2020, pp. 157–168. URL: https://doi.org/10.1109/ICSA47634.2020.00023. doi:10.1109/ICSA47634.2020.00023.

[10] D. Taibi, L. Mezzalira, Micro-frontends: Principles, implementations, and pitfalls, ACM SIGSOFT Software Engineering Notes 47 (2022) 25–29. doi:10.1145/3561846.3561853.

[11] L. Mezzalira, Microfrontends anti-patterns: Seven years in the trenches, 2022. URL: https://www.infoq.com/presentations/microfrontend-antipattern/.

[12] E. Braz, Flutter micro app - a package to speed up the creation of micro frontend(or independent features) structure in flutter applications, 2022. URL: https://web.archive.org/web/20220804142023/https://flutterrepos.com/lib/emanuel-braz-flutter_micro_app.

[13] Ionic, Micro frontend architecture for mobile web apps - ionic portals, 2022. URL: https://ionic.io/portals.

[14] F. Auer, V. Lenarduzzi, M. Felderer, D. Taibi, From monolithic systems to microservices: An assessment framework, Information and Software Technology 137 (2021) 106600. doi:10.1016/j.infsof.2021.106600.

[15] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, M. Mazzara, From monolithic to microservices: An experience report from the banking domain, IEEE Softw. 35 (2018) 50–55. doi:10.1109/MS.2018.2141026.

[16] R. Capuano, H. Muccini, A systematic literature review on migration to microservices: a quality attributes perspective, in: IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022, IEEE, 2022, p. 120–123. URL: https://doi.org/10.1109/ICSA-C54293.2022.00030. doi:10.1109/ICSA-C54293.2022.00030.

[17] F. Freitas, A. Ferreira, J. Cunha, Refactoring java monoliths into executable microservice-based applications, in: C. D. Vasconcellos, K. G. Roggia, P. Bousfield, V. Collereii, J. P. Fernandes, M. Pereira (Eds.), SBLP'21: 25th Brazilian Symposium on Programming Languages, Joinville, Brazil, 27 September 2021 - 1 October 2021, ACM, 2021, p. 100–107. URL: https://doi.org/10.1145/3475061.3475086. doi:10.1145/3475061.3475086.

[18] M. Brito, J. Cunha, J. a. Saraiva, Identification of microservices from monolithic applications through topic modelling, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 1409–1418. URL: https://doi.org/10.1145/3412841.3442016. doi:10.1145/3412841.3442016.