From Monolithic to Microservice Architecture: The Case of Extensible and Domain-Specific IDEs

Romain Belafia, Pierre Jeanjean, Olivier Barais, Gurvan Le Guernic, Benoit Combemale

From Monolithic to Microservice Architecture: The Case of Extensible and Domain-Specific IDEs

Romain Belafia*, Pierre Jeanjean†, Olivier Barais*, Gurvan Le Guernic‡, Benoit Combemale*
*Univ Rennes, Inria, CNRS, IRISA, Rennes, France
†Inria, Univ Rennes, CNRS, IRISA, Rennes, France
‡DGA MI, Bruz, France
{romain.belafia, olivier.barais, benoit.combemale}@irisa.fr
{pierre.jeanjean, gurvan.le_guernic}@inria.fr

***Abstract*—Integrated Development Environments (IDEs) are evolving towards cloud-native applications with the aim to relocate the language services provided by an IDE on distant servers. Existing research works focus on the overall migration process to handle more efficiently their specific requirements. However, the microservicization of legacy monolithic applications is still highly dependent on the specific properties of the application of interest. In this paper, we report our experiment on the microservicization process of the Cloud-Based graphical modeling workbench *Sirius Web*. We aim to identify the technical challenges related to applications with similar properties, and provide insights for practitioners to migrate their similar applications towards microservices. We discuss the main lessons learned and identify the underlying challenges to be further addressed by the community.**

***Index Terms*—Microservice, DevOps, Domain-Specific Languages**

I. Introduction

In order to be fully functional and integrated into their usage environment, software languages need to be delivered with Integrated Development Environments (IDEs). These tools provide facilities to the programmers, such as syntax highlighting or auto-completion. These services, called *Language Services* allow the developer to use these languages more effectively in order to implement new software. Recently, a new trend for IDEs has emerged: Cloud-Based IDEs. These IDEs aim to relocate language services from the user's machine to a distant server. These IDEs have many benefits in comparison to more traditional IDEs, such as automatic saving and backup of code or real time pair programming, without the need to install anything [1]. Protocols such as LSP[1] allowed to standardize the communication between a generic IDE client and a Language Server providing language services. However, the way these servers are deployed, typically as monolithic application, is currently not optimal for such an IDE usage. Indeed, language services have specific requirements in terms of bandwidth or memory [2]. Some need to be accessed frequently, but do not require important computation capabilities, for instance auto-completion, while others such as refactoring are accessed less frequently but need more resources. Besides, modern IDEs tend to move towards an open-world philosophy: in this paradigm, an IDE consists of a basic kernel, which allows the addition of new features incrementally that have access to the same resources, making it more modular. For these reasons we explore a different software architecture more likely to meet these constraints: the Microservice architecture.

Microservices have been recently introduced by the Software Engineering community as a support for new distributed and dynamic software architectures. In contrast to traditional monolithic architectures, a microservice architecture consists of small self-contained lightweight services working together. This new architecture has been widely adopted in the case of complex and heterogeneous projects by companies such as Netflix [3], Spotify [4] or Uber [5]. Indeed, microservices provide many benefits for Cloud-based applications, such as better scalability, autonomy and deployment times [6]. As a result, such architectures are a good fit for DevOps practices [7]. However, the transition from a monolithic to a microservice[2] application is still considered as a long and risky process [9]. Furthermore, even though academic work has been done to provide guides and tools to help companies performing such a modernization, the refactoring of legacy systems remains overall a case-by-case process [10].

This work aims to study the transition process of monolithic applications towards microservices, by focusing on the migration of Domain Specific Cloud-Based IDEs. This case study allows to explore the specific features of such software. These applications offer services to the user, which are heterogeneous in their response time and their computation requirements. They also add an important layer of complexity for the deployment of the services. Last, but not least, these applications manipulate rich data structures which adds challenges for the organization of the microservices.

In this paper we report our experience from the migration towards microservices of Sirius Web[3], an open-source framework allowing the development of Cloud-Based Graphical Language Workbenches. Based on the encountered difficulties, we develop a basis for practitioners to execute the modernization process of systems sharing similar specificities.

This paper is structured as follows: Section II presents the motivations for this work and establishes technical back-

[1]https://microsoft.github.io/language-server-protocol/

[2]In this article, we use the neologism **microservicization**, as in [8], to designate the transition process from monolithic to microservice architecture.

[3]https://www.eclipse.org/sirius/sirius-web.html

grounds on the domains of Software Language Engineering and Microservices. Section III describes the specificities of Cloud-Based IDEs in a microservicization context. Section IV presents the experimental setup of the study, before describing the technologies used. Section V describes the experimentations performed, and the encountered difficulties. Section VI presents the lessons learned in this work: guides on microservicization process for specific industrial cases related to the case study. Section VII reflects on existing and related work to position our contribution. Finally, Section VIII performs a summary of the presented results, before bringing some perspectives for future works.

II. Motivation

A. Software Language Engineering (SLE)

1) Language Implementation: Modeling a language from scratch is a tough process. Indeed, several steps are involved in the creation of a language: first on the syntactic aspect, with the specification of the concrete syntax, the abstract syntax, the static semantics and the dynamic semantics, but also on the behavior that can consist of either compilation or interpretation. Moreover, in order to provide a usable environment for the users, the language has to be delivered with tools that assist its uses. These tools form an *Integrated Development Environment*, or more simply, an *IDE*.

An IDE is a software which assists its user in the development of a program using a particular language. Examples of these are JetBrains CLion[4] for C or C++ development and RStudio[5] for R development. These pieces of software provide *Language Services*: functionalities aiming to assist the developer in writing programs in a particular language. For instance, the auto-completion advises the user to automatically complete a string of characters they are typing, with the most relevant proposition. Compilation can also be considered as a language service which converts source code to object code. Examples of other language services are refactoring, searching for references, etc.

Currently, IDEs tend to move towards an *Open World* philosophy, that is to say the ability to incrementally add functionalities to an existing software. These IDEs such as Microsoft Visual Studio Code[6] or Atom[7] are called modular IDEs. They initially act as simple text editors, that do not offer language specific functionalities. However, these IDEs allow the installation of modules which provide Language Services, for instance the syntax highlighting of C programs or the debugging of Java programs. The trend of modularity in IDEs is very noticeable if we look at the 2019 Stack Overflow's annual Developer Survey[8]. According to this survey, the majority of the most popular IDEs (8 out of 10), such as Visual Studio Code or IntelliJ[9], heavily rely on a plug-in mechanism to customize the environment according to the needs of the users.

With the different steps of specification and the development of an IDE — or modules for modular IDEs — the development of a language can become a fastidious task. In order to facilitate this language development process, tools have been developed: the *Language Workbenches*. Languages workbenches have been introduced and popularized by Martin Fowler in 2005 [11], as environments that can drive the development and the maintenance of Domain Specific Languages (DSLs). Examples of language workbenches are the Eclipse Modeling Framework[10] (EMF) or JetBrains Meta Programming System[11] (MPS). These tools aim to support the development and the evolution of a language. They support every step of the conception of a language, from its design, to the support of its evolution, including the development of the associated IDE [12].

In the end, from the point of view of its implementation, a DSL corresponds to a set of language services provided by an IDE to develop a program [2].

2) Cloud-Based IDEs: Cloud-based IDEs aim to relocate software from the user desktop to a distant server providing the main functionalities of the application, following the Software As A Service (SaaS) model. On the client side remains only a user interface with minimal functions. The client side can be localized either in a browser, or a basic desktop application.

There are a lot of opportunities for Cloud-Based IDEs. To begin with, a SaaS architecture saves the user the trouble of installing heavy and specific software to program such as linters or a JDK [1]: this can possibly allow any user to use the IDE, no matter the resources they have available locally. An important feature of Cloud-based IDEs is the ability to work simultaneously with other coworkers [1], [13], allowing to ease real-time pair programming. Another aspect which has been recently studied is the ability to take advantage of these platforms to analyze the coding habits of either a particular user or a community of users to provide a more relevant assistance, based on their behaviors on the IDE [13], [14]. These explain why Cloud-Based IDEs, such as Jupyter Notebook[12] or Overleaf[13], have gained in popularity.

A Cloud-based IDE works the following way: the client-side is composed of a *language agnostic IDE* (which mostly acts as a simple text editor), and the server-side exposes a *language server* which provides *language services*. Figure 1 illustrates the typical way Cloud Based IDEs are implemented.

The LSP protocol[14], introduced by Microsoft in 2016, standardizes the communication between a language agnostic IDE and a language server. Instead of specifying how each language service of each language works for each IDE, the

[4]https://www.jetbrains.com/clion/
[5]https://rstudio.com/
[6]https://code.visualstudio.com/
[7]https://atom.io/
[8]https://insights.stackoverflow.com/survey/2019#development-environments-and-tools
[9]https://www.jetbrains.com/idea/
[10]https://www.eclipse.org/modeling/emf/
[11]https://www.jetbrains.com/mps/
[12]https://jupyter.org
[13]https://www.overleaf.com/
[14]https://microsoft.github.io/language-server-protocol/

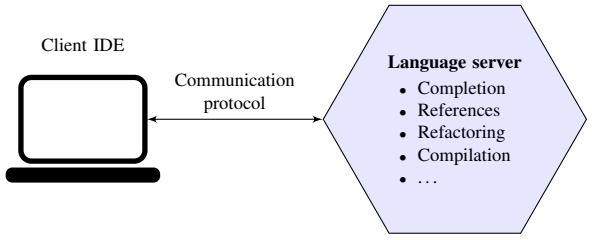

Fig. 1: Legacy implementation of Cloud-based IDEs

LSP protocol allows to standardize the way these services are rendered to the user. This protocol was originally developed for the creation of modules for Microsoft Visual Studio Code, but has become a standard for other environments as well. Through its use, a single language server implementation can be used as-is in a landscape of development environments, and any environment can leverage on the existence of multiple language servers. Modular IDEs also allow their users to personalize their own user experience by choosing the functionalities they judge necessary.

However, current Cloud-Based DSL IDEs are still generally implemented in a monolithic way [2]. All the services are rendered by one non-modular application. In addition to the lack of customization of these IDEs, the language services are not optimal in their response time. Indeed, each service has specific requirements in terms of bandwidth or latency, but they are all still deployed on the same hardware. Another way to implement such a language server has been explored by Coulon et al. [2]. This new approach is based on a microservice architecture, which is a deployment organization that separates every functionality of the system as independent services.

Cloud-Based DSL IDEs have a lot to gain to transition towards microservices. The different language services provided by the IDE have different requirements in terms of bandwidth or latency. Moreover, IDEs tends to move towards an open-world philosophy, by separating the language services in clear modules which can be added to the IDE incrementally. Such requirements could be more easily met in a microservice architecture rather than in a monolithic one.

B. Microservice Architectures

1) Definition and Properties of Microservice Architectures: Microservice architecture is an approach of software service design, development and delivery, which has gained a lot of popularity in the recent years. It has been widely used in the industry, more particularly in the case of large-scale applications. For instance, companies such as Netflix [3], Spotify [4] and Uber [5] have been using it as the basis for complex infrastructures with great success.

The main characteristic of a microservice architecture is the decomposition of the system in clear modules. These modules, the microservices, are lightweight and independent. They support a single functionality in the program the best way possible [15]. Therefore, such a service would typically only be a hundred to a thousand lines of code long [16]. Each service is deployed independently, and depending on its specific needs, a lightweight service can be deployed locally while a heavier service, needing more resources but not a fast response time, can be deployed on a remote server. These services communicate between each other using a Language Agnostic API [17]. Microservices are generally packaged and deployed in the cloud using lightweight container technologies such as Docker[15], following the fashion of DevOps practices. These architectures are typically supported by fully automated software integration and delivery machinery such as Gitlab[16] and Jenkins[17].

Such architectures are currently becoming a standard in software engineering. In [7], the authors emphasize the benefits that microservices architectures can bring to DevOps practices. For instance, microservices are known for their ability to deal with scalability [6], i.e. the ability to make the size of the system evolve as the project evolves. This ability is crucial in a DevOps-driven development, to quickly adapt a system to its usage. In addition to such technical benefits, microservices also allow for more process-related benefits, such as the distribution of the workforce into small and self-managed teams, each focused on a microservice. This fits very well DevOps practices which recommend to vertically divide the workforce into small cross functional teams [18].

2) Transition Process from Legacy Systems Towards a Microservice Architecture: Here, we discuss the modernization process, which is undertaken by industrials when performing a transition from legacy systems towards microservice architectures. In [19], the authors establish 4 phases to transition monolithic applications towards microservices, based on a body of academic works studying the microservicization process.

a) Analysis of the Driving Forces: First, the analysis of the driving forces is considered as a crucial step of a modernization. Indeed, while the transition to microservice can bring significant improvements to the application in the long term, and ease future developments, transition from legacy systems towards microservice architecture is a long and risky process [9]. Thus, companies often establish cost-benefit analysis, based on the potential benefits such as increased performances, better scalability and ease of the development process [6].

b) Modernization Planning: The next phase of a modernization is the establishment of a plan for the transition towards microservices. To begin with, practitioners explore, analyze and understand the legacy systems: the different features, how they interact, how they are implemented. This process is called *feature location*. In order to accomplish this task, practitioners can rely on high-level artifacts such as UML diagrams, graphs, or texts based on low-level artifacts such as source code [20]. The next step is to decompose the legacy

[15]https://www.docker.com/
[16]https://about.gitlab.com/
[17]https://www.jenkins.io

system in small modules which will then be implemented as microservices. This very complex step has been investigated by a lot of research works. Here, the practitioners decide on the granularity of the decomposition, that is to say the balance between the number of microservices and their size. Finally, the microservice architecture is properly defined, by describing the microservices behavior and the way they communicate. This is done by defining the services' APIs and the communication protocols, such as HTTP or SOAP, used for the exchanges between microservices [6]. However, the decomposition of legacy systems remains in most cases a manual process. Indeed, the microservices are often designed around business capabilities, to allow their independent deployment [21]. These capabilities are not easily identifiable through an automatic process.

c) Modernization Execution: Once the modernization has been planned, the next step is to execute this plan, by developing and integrating the microservices. A few research works aimed at automating the modernization execution. For instance, in [22] the authors developed a Model-Driven approach which automates the transition from a Java based monolithic application to a microservices-based application. However, here again, such tools are rarely used in the industry, and the implementation task remains mostly manual. In parallel, or after the development of these microservices, comes the test phases. While automated unit tests or integration tests are usually done alongside the implementation, some validation tests can still come afterwards.

d) Monitoring: Based on metrics defined upstream, the microservicized application needs to be monitored in order to verify that the aims have been reached. Tools have been developed to assist this process such as Micrometer[18] which provides interfaces to evaluate JVM-based applications.

Our work aims to explore the specific case of Cloud-Native IDEs, as a way to analyze the constraints linked to the manipulation of rich data structures, in the case of a migration towards microservice architectures. The next section describes in more details this case study.

III. On IDE Microservicization

As shown in Section II-A2, Cloud-Based IDEs have a lot to gain to be migrated from a monolithic to a microservice architecture. Besides, IDEs are interesting case studies due to their particularities as software.

A. Heterogeneous Services

As explained in Section II-A1, an IDE provides multiple language services (auto-completion, compilation, refactoring, etc.). In a microservices approach, each one of these language services would correspond to a single microservice. An important aspect of these services is their heterogeneity, whether they have different access frequencies or required resources. For instance, auto-completion aims to propose to the user to automatically complete a string of characters they are typing. Such a service is accessed very often — almost at each user input. However, such a service usually does not require a lot of computation power, in comparison to other services. Conversely, compilation is usually a heavy process. The compilation of a C program can take in some cases tens of minutes. Nevertheless, such a time-consuming process is usually accepted since compilation is called far less frequently than auto-completion for instance. Such a diversity of services often means a diversity of services implementations, but also of deployments.

B. Complexity of Deployment

The question of microservices deployment and more generally of cloud deployment is complex due to the multiple factors taken into account in addition to the response time [23]. Elements such as availability, free storage, CPU usage or memory usage are constraints that are imposed to the deployment centers, which have specific needs [24]. In the case of Domain-Specific IDEs, the question of services deployment is even more complex due to the volatility of the resources. Depending on the usage of the IDE, the optimal deployment for the microservices differs. For instance, if a user is the only one to use a given instance, the deployment will be different from when a 10-person team is fully committed to a project. Furthermore, in the last case, the resources available for the deployment are susceptible to be unavailable at a certain point. The deployment mechanism must thus adapt more or less dynamically to the resources currently available. The *black box* aspect of the services is also a challenge. Indeed, the resource consumption of the services can be difficult to evaluate before the deployment, and the resources needed are often only known once the application begins to run.

C. Rich and Shared Data Structures

The goal of an IDE is to provide assistance to the user to write a program. A program is manipulated as a rich data structure, usually in the form of an Abstract Syntax Tree (AST). These structures have to be accessible by language services, more or less frequently depending on the services, as explained previously. Therefore, the transfer and the serialization of these structures have to be handled properly to avoid any drop in performances. If the entirety of the program and the associated resources were transferred at every call of a service such as auto-completion, this could have a great impact on the performances and make the overall environment feel *sluggish*. Thus, one has to make choices, both about the structures of the microservices and the data to transfer between them. For instance, we can imagine a system that keeps a copy of the AST in every microservice, which would be modified on each call through diffs and patches to stay coherent with the other services.

IV. Experimental Setup

A. Case Study Overview

In order to study the modernization of Cloud-Native IDEs, we rely on Sirius Web by Obeo. Sirius Web is a framework

[18]https://micrometer.io/

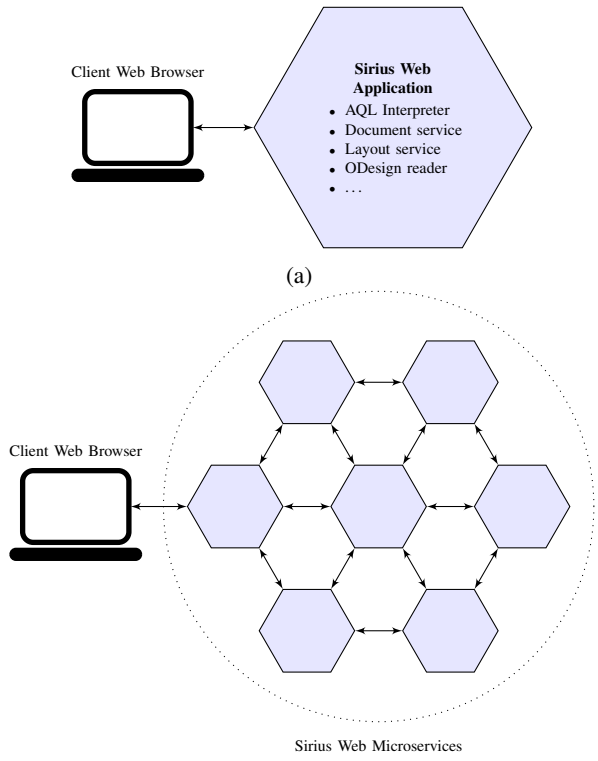

Fig. 2: Representation of a Sirius Web application structure: (a) the legacy monolithic structure (b) the new microservice structure we aim for

dedicated to the creation of *Graphical Modeling Workbenches*. A graphical modeling workbench can be described as an IDE that manipulates graphical modeling languages. Sirius Web is mainly used in the context of model engineering, an approach which relies heavily on models instead of textual code as the source to generate software.

The first implementation of Sirius was a desktop application, integrated as an Eclipse application. Examples of Sirius usages are Capella[19] by Thales, dedicated to the development of system architectures, or AdvoCATE[20] by NASA, used to specify safety and assurance cases. Recently, Sirius has been ported to the Cloud, as a monolithic application.

Here, we study the transition process of Sirius Web from a monolithic application to microservices. The aim of this work is to study the modernization process of Sirius, in order to highlight some difficulties that such architectures can cause. Figure 2 represents the legacy architecture of Sirius Web and the modern microservice architecture we aim for.

B. Technologies Overview

1) Eclipse Modeling Framework & Meta-models: To describe a meta-model, Sirius uses the Eclipse Modeling Framework[21] (EMF), a set of Eclipse plug-ins which can be used in the practice of *modeling engineering*. EMF provides a tool, Ecore, to describe graphically a meta-model. Ecore is a framework composed of a set of *concepts*, that can be manipulated by EMF to build a meta-model. In addition, EMF allows specifying, in a *genmodel* file, the generation of Java code from an Ecore meta-model, which will be then manipulated by Sirius.

2) Eclipse Sirius: Once the meta-model of the manipulated model has been generated, Sirius allows the user to define its graphical representation. On top of an EMF Project, Sirius allows the creation of an *odesign* file, to specify the graphical representation of the different concepts described with Ecore. For instance, choosing icons to represent the instances of an EClass, or the EReferences with straight lines.

Sirius also offers the possibility to specify conditional rules, for the representation of models. For instance, an element could appear bigger if a numeric attribute x is superior to 10. To specify these conditions, Sirius uses a query language, the Acceleo Query Language[22] (AQL). AQL is a Domain-Specific Language (DSL) developed by Eclipse, to perform queries over an Ecore meta-model. If a behavior is too complex to be specified using AQL, it can be described by adding additional Java code through what is referred to as "Java Services".

The original version of Sirius, Sirius Desktop, allows to produce an RCP application, but recently Obeo released a Cloud-Native version of Sirius: Sirius Web. Sirius Web allows the development of a Web Application running in a Web Browser.

A Sirius Web application relies on a set of modules called *Sirius Components*[23]. In this work, we focus in particular on the backend components managing the back-end part of Sirius Web. They are implemented using Java, and rely on the Spring Boot framework[24]: a Java framework, particularly adapted for the creation of Web Applications. Each component provides a particular functionality of Sirius Web or describes their associated API. Other Sirius Components are dedicated to testing the behaviors of a set of Components designed to work together.

V. Experimentation

A. Feature Location & Analysis of Dependencies

To begin with, the first step to migrate Sirius Web towards microservices is to analyze the different functionalities of the legacy system. The idea being to locate features which could lead to a specific microservice. As explained in Section IV-B2, Sirius Web relies on Sirius Components, which are the building blocks of any Sirius Web application. This decomposition is an example of code modularity. It tends to ease the identification of features inside the application, and make the overall structure easier to understand.

However, the current decomposition has some limitations. Indeed, this code modularity does not necessarily correspond

[19]https://www.eclipse.org/capella/
[20]https://ti.arc.nasa.gov/tech/rse/research/advocate/
[21]https://www.eclipse.org/modeling/emf/
[22]https://www.eclipse.org/acceleo/documentation/
[23]https://github.com/eclipse-sirius/sirius-components
[24]https://spring.io/

to the deployment modularity we aim for: each component does not correspond directly to a specific functionality, which could correspond to a microservice. For instance, some components only provide the API of the system while other specify some annotations used in others. Features to isolate are often features contained inside Sirius Components.

An important part in this process is the identification of dependencies between the different features to isolate. Here again, the modular structure of the Sirius Components allows to easily identify them since each Sirius Component is built as a Maven[25] project. Thus, all the dependencies of a Sirius component are specified in the corresponding `pom.xml` file.

Identifying dependencies between components allows to determine the decomposition in microservices and the overall architecture. The components which have the fewer dependencies on others will be easier to extract. Based on this study we plan the decomposition of the application in different services. The identified features to isolate consist of:

- The AQL Interpreter, performing AQL queries of the meta-model;
- The Document Service, dealing with create, read, update and delete (CRUD) operations;
- The Layout Service, organizing the layout of the representations;
- The ODesign Reader, which aims to compute an odesign file.

The interactions between the different features of the application being very complex, we decided to perform the decomposition of the application incrementally. That is to say, we decided to plan and execute the decomposition of one microservice at a time, rather than planning the microservicization of all the features identified, before executing their modularization. This approach is not specific to this work. Indeed, some academic research such as [25] argue that the microservicization process of an application should be incremental. Due to the complexity of the activity, one will slowly learn how to better perform this decomposition for each microservice successfully extracted from the monolithic system. Thus, this organization would help us to better apprehend the microservicization process at each increment.

For each component extracted from the rest of the application, we pay attention to the difficulties encountered, and how they can be generalized to other applications. This will allow us to derive practice guides for the modernization of a monolithic application.

B. AQL Interpreter Decomposition

The first feature chosen for the decomposition of Sirius Web in microservices is the Acceleo Query Language Interpreter — or AQL Interpreter. This part of the application aims to interpret AQL expressions, which are queries performed on EMF models. The reason for this choice is that the AQL Interpreter does not have any dependencies on other Sirius components, which tends to make the decomposition in microservices easier. Moreover, an AQL Interpreter microservice could be reused in future Cloud-Based IDEs using the Eclipse Modeling Framework.

1) Original Implementation: The AQL interpreter was fully implemented in the `sirius-web-interpreter` component. Before evaluating an AQL expression, an `AQLInterpreter` needs to be initialized. To do so, one must pass two arguments:

- A set of Java classes, corresponding to the Java services available to the AQL interpreter;
- A set of EPackages, the Java code generated for additional meta-models which can be interpreted by the AQL interpreter.

Once an AQL interpreter has been initialized, it can start evaluating AQL expressions. Here, two elements are required:

- A set of variables, that the AQL evaluator can use;
- The AQL expression to evaluate.

The result of an evaluation can be an instance of various types: a String, an Integer, an Object, etc. This result is wrapped in an instance of `Result`. A `Result` encapsulates the result of the expression, and allows delaying the typing of the resulting value.

AQL interpreters are used in other components. The original implementation of the AQL interpreter consisted in many instances of `AQLInterpreter` used in the various components. The idea of the microservicization was instead to have only one AQL interpreter service which could be called by other components.

2) Microservice Architecture: In order to explore the different ways to perform microservice migration, two implementation approaches were explored: a stateless and a stateful approach.

a) Stateless Approach: In accordance to the current trends of microservices, the first approach investigated was a *Stateless* approach. This means that the AQL interpreter should not retain any information from previous interactions. Thus, no initialization phase is done in this implementation. Each time an AQL expression is evaluated, the Java services and additional metamodels needed are passed to the AQL interpreter. This approach has the typical advantages of Stateless architectures: the on-demand elasticity and the reliability through redundancy. Moreover, an AQL expression can be directly evaluated without the need for an initialization phase. However, this approach turns out to be very cumbersome for the rest of the application. Indeed, the Java services and additional packages must be carried between the components. If this can be handled in a monolithic approach, it can quickly become an issue in a microservice architecture. Moreover, since no initialization phase is done with the AQL interpreter, a similar work has to be done at each evaluation. This can result in an important drop in performances, similar to the one observed in [2].

b) Stateful Approach: The *Stateful* approach has also been investigated. In this approach, the initialization phase is

[25]https://maven.apache.org/

preserved. Thus, before performing an evaluation, one must call the initialization function of the AQL microservice. A "session" will be created as a `IQueryEnvironment`. This corresponds to the information built during the initialization of the Interpreter based on Java services and additional meta-models. Each session is identified with an integer (more precisely, a `Long`, to avoid running out of session IDs). Thus, each time an evaluation has to be done, the session ID has to be passed in order to evaluate the expression in the right environment. This approach loses the advantages of statelessness, in particular the reliability through redundancy: the same call to the AQL interpreter will not always produce the same output. However, it allows to make the manipulation of the AQL interpreter less complex. Instead of manipulating an instance of `AQLInterpreter` like the original implementation, or a set of Java classes and EPackages like the Stateless approach, here a simple integer is enough to communicate with the AQL Interpreter. For these reasons the Stateful approach has been selected.

3) New Implementation: The new AQL interpreter microservice has been developed using the Quarkus Framework[26]. This Java framework, similar to Spring Boot, is here again particularly suitable for the development of Cloud applications, and microservice applications in particular. Quarkus consists of a set of technologies optimized for GraalVM[27]. GraalVM is a virtual machine considered as "universal". It allows running applications written in many languages: JVM-based languages such as Java, Scala or Kotlin, LLVM-based languages such as C and C++, and other dynamic languages such as JavaScript, Python, R or Ruby. GraalVM makes possible for Quarkus to perform ahead-of-time (AOT) compilation, converting the produced Java bytecode into native machine code, resulting in an optimized binary that can be executed natively. This allows to reduce considerably the start-up time and the memory footprint of the application, compared to similar software made with Spring-boot [26].

For communicating with the microservices, we decided to use a RESTful implementation. This was implemented using the RESTEasy Framework[28] by JBoss. This framework provides an implementation of the JAX-RS API, a Java API for RESTful services.

C. Next Components

Unfortunately, the decomposition of the AQL Interpreter was a very complex process which did not allow us to focus on other services. In future works, the decomposition of other components will be explored. For instance, the Document Service, dealing with CRUD operations, will allow to explore the decomposition of a service dealing with a database. The ODesign reader, which relies on the AQL Interpreter, can serve as a case study to explore the communications between differents microservices.

[26]https://quarkus.io/
[27]https://www.graalvm.org/
[28]https://resteasy.github.io/

However, the difficulties encountered during the decomposition of the AQL Interpreter allowed us to derive practice guides, that we will talk about in the next Section.

VI. Lessons Learned

This section discusses the difficulties encountered during the microservicization of Sirius Web, and the lessons learned for future applications. We describe broader contexts where such difficulties can be encountered, and discuss the possible solutions which can be applied.

A. Serialization Issues

A difficulty encountered during the microservicization of Sirius concerned the serialization process. The serialization allows converting Java objects to streams of bytes, in order to easily transfer these data between services — typically using json or XML formats. In the case of complex data structures such as Abstract Syntax Tree, serialization can be a tough process. Indeed, elements such as inheritance of concepts defined in external projects or self-references can add a layer of complexity to the process. Nevertheless, in the case of Sirius, some tools already existed to serialize EObjects. The basic way to serialize EMF objects consisted in using the XML Metadata Interchange[29] (XMI) standard. This standard allows specifying the representation of meta-data in an XML format. More recently, the emf-json project[30] provided a support for the JSON format to the Eclipse Modeling Framework. However, even though these formats were adapted to transfer the EMF instances found in models, they were not designed to deal with the EMF classes specifying the meta-models.

1) Application Cases: Such an issue can be encountered in a lot of situations, as soon as the elements to exchange are not trivial to serialize. The presence of rich and complex data structures to transfer can be an evidence for such issues.

2) Practice Guide: In the planning phase, practitioners aim to plan the decomposition of the application. It is important at this point to anticipate which data must be transferred between the services, and how they will be serialized. In the case of uncommon data structures, practitioners have to be aware of whether dedicated tools already exist for the serialization of these structures or not. If this is not the case, they must anticipate the difficulties that will be encountered and decide on the way to resolve them before attempting any decomposition.

Lesson Learned 1 Serialization of data to transfer can be an arduous process when rich and complex structures are manipulated. These difficulties should be planned upstream, especially to audit existing solutions or to schedule the development of a new one.

[29]https://www.omg.org/spec/XMI
[30]http://emfjson.github.io/

B. Rupture of Pass-by-Reference Chains

In a program, a method or a function is characterized by the elements passed as actual parameters. The way these elements are passed can be grouped up in two main categories:

- In a **pass by value** evaluation strategy, the parameters are passed as a copy of the object passed to the function, which will prevent the original object from being affected by the operations done by the function;
- In a **pass by reference** evaluation strategy, a reference to the object is passed, which allows modifying directly the element pointed by the reference.

In a Java program, the evaluation strategy is called *Shared Object Transfer*, and is in practice similar to the pass-by-reference strategy. It allows to easily manipulate Java objects passed as the actual parameters of successive methods or functions, creating pass-by-reference chains. However, this way of manipulating data can cause issues in a microservicization context. Indeed, when a feature is separated from the legacy application to its own microservice, the communication by pass-by-reference is cut during the serialization process. Basically, this means that every object passed to a microservice is in fact "passed by value", breaking up existing pass-by-reference chains.

1) Application Cases: If the legacy applications relies heavily on a pass-by-reference strategy, such problems can occur. In particular, such issues are more likely to occur if the possibility to alter a function parameter is not made explicit.

2) Practice Guide: During the analysis of the legacy systems, practitioners must take note of the way variables are transferred throughout the program. If the application is planned for a migration toward microservices, one would prefer a more functional style. Otherwise, practitioners must meticulously adapt the structure of the application. The called microservice has to apply any changes on function parameters and notify its caller afterwards.

Some future works could explore the development of a framework that allows to explicitly specify how a parameter should be treated during the microservicization. For instance, in the case of a Java program, an annotation framework could be developed to indicate if a parameter is expected to be affected by side effects or not. This could allow developers to analyze with more correctness the feature map in a program, but could also open the way for microservicization processes based on these annotations.

Lesson Learned 2 An application which heavily relies on pass-by-reference chains exposes itself to issues during the modernization process. One must thus be vigilant to adapt these behaviors to the future microservices' interactions.

C. Open-World Challenges

Sirius Web relies on the open-world philosophy. Indeed, it allows the enrichment of the application through the addition of new functionalities, in particular using Java Services. This philosophy makes a lot of sense in the context of IDEs, since it allows a user to add functionalities incrementally to fit its use-cases, as many other modular IDEs do. However, this makes the microservicization of the application much more complex. Indeed, multiple microservices must often have access to these added functionalities.

In the case of the AQL interpreter, the problem came from the presence of Java Services required during the initialization phase. However, the microservice architecture prompts the practitioners to develop the microservices to be deployable and usable independently of their usage context. Thus, the microservices need to dynamically adapt to these functionalities.

1) Application Cases: Such problems can be encountered in open-world applications, i.e. applications offering the possibility to dynamically add new functionalities. The closest example to the case study is the case of modular IDEs, but for instance one can also imagine a cloud-based video game which offers the possibility to manage additional mods.

2) Practice Guide: In order to solve these issues, the first element to take into account is the possibility for the technology to adapt to dynamic functionalities. For instance, the AQL interpreter was implemented using Quarkus, which offers the possibility to run native applications, lowering both the memory footprint and the startup time. As a downside, such an approach prevents the deployed microservices to adapt to modular functionalities. However, other technologies allow for a more dynamic behavior. We thus recommend anticipating such problems at the earliest, particularly when deciding on the microservice technologies.

Another choice to handle open-world issues is to better define the granularity. Indeed, the ability to manage modular functionalities can be handled by using dedicated microservices. However, such an implementation is not trivial either: microservices have to coordinate to adapt to this new functionality. New architectural challenges appear to deal with this structure.

Lesson Learned 3 In an open-world application, the availability of modular features for different microservices can be an issue. Practitioners must plan how to make microservices deal with the dynamic addition of features, possibly by taking them into account when considering the granularity of the application.

D. Manipulation of Shared Resources

In some application cases, multiple features can rely on shared resources: heavy data which can be initialized once, and then accessed by different functionalities of the program. For instance, the AQL interpreter, as well as other features of Sirius Web, relies on additional EPackages to initialize an interpreter. The fact that a monolithic application may decide to share the same resources with different features of the application makes sense, and allows mechanisms such as lazy

initialization. However, their usage in the case of microservices raises some issues.

As explained in Section VI-A, these issues can be linked to the serialization of these resources or the synchronization between the different components. In addition to these, the horizontal scaling of microservices implies the creation of multiple instances of the same resources in different services, causing higher costs in memory footprint and greater instantiation times.

1) Application Cases: Applications sharing rich resources throughout multiple different features are subject to such challenges.

2) Practice Guide: In the same way as the previous issue, a solution to deal with such a challenge is to carefully plan the microservicization architecture. Centralizing the services which manipulate shared resources can solve the issue. However, such a solution is not always possible and does not fit well with the philosophy of microservices. Another research lead to investigate is the possibility to share such resources locally. Once again, such a solution is difficult to implement and adds additional constraints to the deployment strategy of the application.

Lesson Learned 4 Shared resources in an application raise challenges for the microservicization process. In such cases, one must be aware of the impacts of such elements on the microservice granularity and the overall deployment.

VII. Related Works

The process of Cloud-based IDE microservicization has already been explored in [2]. This article develops a systematic approach to migrate monolithic Cloud-Based IDEs towards microservices. Based on the specification of the manipulated language and the desired language services, the approach generates a set of modular language microservices and a tool-supported feature model to configure their deployment. This approach brings disparity in the performances of language services after their modularization. If heavy services such as compilation show improvements on their response time from this approach, lightweight quick-feedback services such as auto-completion suffer greatly in reactivity. This drop in performances is even more accentuated when the length of the code increases. The authors of the paper have already identified two factors as responsible for this disparity: the statelessness of the services and the lack of optimization for the deployment.

Some research works have been done to automate the decomposition of a Cloud-based application, based on the expected benefits for the modernization. For instance, Service Cutter relies on an Entity-Relationship model to propose a decomposition of a legacy application [27]. In [22], the authors developed a Model-Driven approach which automates the transition from a Java based monolithic application to a microservice application. This approach is based on *Jolie*[31], a programming language designed to implement microservices and specify microservice architectures. Some academic works established a broader classification on microservicization techniques. In [28], the authors performed a "rapid review" on microservicization techniques and highlighted 3 main approaches to perform such a modernization : model-driven based on high level representations of the system, static analysis of the source code, and dynamic analysis of the running system. Similarly, in [10], the authors identified 10 architectural refactoring approaches for microservicization, classified in similar categories. This work is accompanied by a selection guide to support the practitioners in choosing the more adequate solution.

VIII. Conclusion & Perspectives

We detail in this paper our experience on the microservicization of a Cloud-based IDE. This work aims to provide insights for developers on the difficulties that such applications can raise during a modernization process. These difficulties come from the various specificities that Cloud-based IDEs possess, such as the heterogeneity of the language services' needs or the manipulation of rich and complex data structures. To explore these specificities, we rely on a Graphical Modeling Workbench, *Sirius Web*: a framework that allows users to deploy Cloud-based IDEs manipulating graphical languages.

This paper focused mainly on the microservicization of the AQL interpreter. This part of the application aims to interpret AQL expressions, which are queries performed on an EMF meta-model. The reason for this choice is that the AQL Interpreter does not have any dependencies over other Sirius components, which tends to make the decomposition in microservices easier. This microservicization allowed identifying four main challenges that a Cloud-based IDE can raise in a microservicization process:

- The serialization issues;
- The rupture of pass-by-reference chains;
- The open-world challenges;
- The manipulation of shared resources.

This paper opens the way for future works. To begin with, this experience report focused on the microservicization of Sirius Web's AQL Interpreter. However, other components could be interesting to extract. For instance, the *DocumentService*, in charge of CRUD operations would allow to explore the manipulation of these operations on a Cloud-Native IDE context. *DiagramDescriptionConverter* could also be an interesting case study to explore the manipulation of other types of resources. More generally, the decomposition of Sirius Web in many microservices could be interesting to tackle issues linked to microservice deployment.

Finally, the lessons learned here open the way for the development of microservicization automation tools. For instance, solutions to deal with the rupture of pass-by-references chains could be explored. One could develop an annotation

[31] https://www.jolie-lang.org/

framework to specify the way a passed-by-reference parameter should be handled during the modularization. However, these automatic migration strategies should be diverse, to adapt to different use cases. Each strategy would have strengths and weaknesses. For instance, deciding to use a finer granularity could allow dealing efficiently with applications relying on an open-world strategy, but would add additional costs for an application manipulating shared resources.

REFERENCES

[1] L. M. Gadhikar, L. Mohan, M. Chaudhari, P. Sawant, and Y. Bhusara, *Browser Based IDE to Code in the Cloud*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 59–69. [Online]. Available: https://doi.org/10.1007/978-3-642-35461-8_6

[2] F. Coulon, A. Auvolat, B. Combemale, Y.-D. Bromberg, F. Taïani, O. Barais, and N. Plouzeau, "Modular and Distributed IDE," in *SLE 2020 - 13th ACM SIGPLAN International Conference on Software Language Engineering*, Virtual, United States, Nov. 2020, pp. 1–13. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02964806

[3] E. Bukoski, B. Moyles, and M. McGarr, "How we build code at netflix," 2016. [Online]. Available: http://techblog.netflix.com/2016/03/how-we-build-code-at-netflix.html

[4] S. Engineering, "How we use backstage at spotify," 2020. [Online]. Available: https://engineering.atspotify.com/2020/04/21/how-we-use-backstage-at-spotify/

[5] E. Haddad, "Service-oriented architecture: Scaling the uber engineering codebase as we grow," 2015. [Online]. Available: https://eng.uber.com/service-oriented-architecture/

[6] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[7] M. Waseem, P. Liang, and M. Shahin, "A systematic mapping study on microservices architecture in devops," *Journal of Systems and Software*, vol. 170, p. 110798, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302053

[8] P. Jeanjean, B. Combemale, and O. Barais, *IDE as Code: Reifying Language Protocols as First-Class Citizens*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3452383.3452406

[9] R. Seacord, D. Plakosh, and G. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Professional, 02 2003.

[10] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *DEVOPS*, 2018.

[11] M. Fowler, "Language workbenches: The killer-app for domain specific languages?" June 2005. [Online]. Available: http://martinfowler.com/articles/languageWorkbench.html

[12] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. R. Steel, and D. Vojtisek, *Engineering Modeling Languages*. Chapman and Hall/CRC, Nov. 2016. [Online]. Available: https://hal.inria.fr/hal-01355374

[13] Y. Wang, P. Wagstrom, E. Duesterwald, and D. Redmiles, "New opportunities for extracting insights from cloud based ides," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 408–411. [Online]. Available: https://doi.org/10.1145/2591062.2591105

[14] Y. Wang, "Characterizing developer behavior in cloud based ides," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2017, pp. 48–57. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ESEM.2017.27

[15] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[16] J. Thones, "Microservices," *IEEE Software*, vol. 32, pp. 116–116, 01 2015.

[18] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[19] D. Wolfart, W. K. G. Assunção, I. F. da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, and D. d. N. Paza, *Modernizing Legacy Systems with Microservices: A Roadmap*. New York, NY, USA: Association for Computing Machinery, 2021, p. 149–159. [Online]. Available: https://doi.org/10.1145/3463274.3463334

[20] T. Erl, *SOA Design Patterns*, 1st ed. USA: Prentice Hall PTR, 2009.

[21] A. Levcovitz, R. Terra, and M. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," *ArXiv*, vol. abs/1605.03175, 05 2016.

[22] A. Bucchiarone, K. Soysal, and C. Guidi, *A Model-Driven Approach Towards Automatic Migration to Microservices*. Cham: Springer International Publishing, 2020, pp. 15–36.

[23] W. Ma, R. Wang, Y. Gu, Q. Meng, H. Huang, S. Deng, and Y. Wu, "Multi-objective microservice deployment optimization via a knowledge-driven evolutionary algorithm," *Complex & Intelligent Systems*, Aug 2020. [Online]. Available: https://doi.org/10.1007/s40747-020-00180-1

[24] E. Fadda, P. Plebani, and M. Vitali, "Monitoring-aware optimal deployment for applications based on microservices," *IEEE Transactions on Services Computing*, vol. PP, pp. 1–1, 04 2019.

[25] S. Hassan, N. Ali, and R. Bahsoon, "Microservice ambients: An architectural meta-modelling approach for microservice granularity," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 1–10.

[26] S. Decanay, "Quarkus vs. spring," 2020. [Online]. Available: https://www.logicmonitor.com/blog/quarkus-vs-spring

[27] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing*, M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, Eds. Cham: Springer International Publishing, 2016, pp. 185–200.

[28] F. Ponce Mella, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," 09 2019.