The Collaborative Modularization and Reengineering Approach *CORAL* for Open Source Research Software

Christian Zirkelbach
Software Engineering Group
Kiel University
Kiel, Germany
czi@informatik.uni-kiel.de

Alexander Krause
Software Engineering Group
Kiel University
Kiel, Germany
akr@informatik.uni-kiel.de

Wilhelm Hasselbring
Software Engineering Group
Kiel University
Kiel, Germany
wha@informatik.uni-kiel.de

***Abstract*—Software systems evolve over their lifetime. Changing requirements make it inevitable for developers to modify and extend the underlying code base. Especially in the context of open source software where everybody can contribute, requirements can change over time and new user groups may be addressed. In particular, research software is often not structured with a maintainable and extensible architecture. In combination with obsolescent technologies, this is a challenging task for new developers, especially, when students are involved. In this paper, we report on the modularization process and architecture of our open source research project *ExplorViz* towards a microservice architecture. The new architecture facilitates a collaborative development process for both researchers and students. We explain our employed iterative modularization and reengineering approach *CORAL*, applied measures, and describe how we solved occurring issues and enhanced our development process. Afterwards, we illustrate the application of our modularization approach and present the modernized, extensible software system architecture and highlight the improved collaborative development process. After the first iteration of the process, we present a proof-of-concept implementation featuring several developed extensions in terms of architecture and extensibility. After conducting the second iteration, we achieved a first version of a microservice architecture and an improved development process with room for improvement, especially regarding service decoupling. Finally, as a result of the third iteration, we illustrate our improved implementation and development process representing an entire, separately deployable, microservice architecture.**

Keywords–collaborative software engineering; software modularization; software modernization; open source software; microservices.

I. Introduction

Software systems are continuously evolving during their lifetime. Changing contexts, legal, or requirement changes such as customer requests make it inevitable for developers to perform modifications of existing software systems. Open source software is based on the open source model, which addresses a decentralized and collaborative software development. In this paper, we report on the iterative modularization process of our open source research project *ExplorViz* towards a more collaboration-oriented development process featuring a microservice architecture based on our previous work [1].

Open research software [2] is available to the public and enables anyone to copy, modify, and redistribute the underlying source code. In this context, where anyone can contribute code or feature requests, requirements can change over time and new user groups may appear. Although this development approach features a lot of collaboration and freedom, the resulting software does not necessarily constitute a maintainable and extensible underlying architecture. Additionally, employed technologies and frameworks can become obsolescent or are not updated anymore. In particular, research software is often not structured with a maintainable and extensible architecture [3]. This causes a challenging task for developers during the development, especially when inexperienced collaborators like students are involved. Based on several drivers, like technical issues or occurring organization problems, many research and industrial projects need to move their applications to other programming languages, frameworks, or even architectures. Currently, a tremendous movement in research and industry constitutes a migration or even modernization towards a microservice architecture, caused by promised benefits like scalability, agility, and reliability [4]. Unfortunately, the process of moving towards a microservice-based architecture is difficult, because there a several challenges to address from both technical and organizational perspectives [5]. We later call the outdated version *ExplorViz Legacy*, and the new version just *ExplorViz*. Our main contributions in this paper are:

- Identification of technical and organizational problems in our monolithic open source research project *ExplorViz*.
- An iterative modularization and reengineering process focusing on collaborative development applied on our project moving towards a microservice architecture in three iterations.
- A proof-of-concept implementation, followed by an evaluation based on several developed extensions, as the result of the first iteration.
- An improved software architecture based on microservices and development process after our second iteration.
- Finally, after our third iteration, an entire and separately deployable microservice architecture.

The remainder of this paper is organized as follows. In Section II, we illustrate our problems and drivers for a modularization and architectural modernization. Afterwards, we present the initial state our software system and underlying architecture of *ExplorViz Legacy* in Section III. Our employed modularization and modernization process as explained in Section IV. The following first iteration of this process as well as the target architecture of *ExplorViz* are described in Section V. Section VI concludes the first iteration with a proof-of-concept implementation in detail, including an evaluation based on several developed extensions. The second iteration of our process in terms of achieving a first microservice architecture is presented in Section VII. As there was still room for improvement, we describe how we further improved our microservice architecture and development process in Section VIII. Section IX discusses related work on modularization and modernization towards microservice architectures. Finally, the conclusions are drawn, which includes a summary, depicts lessons learned, and gives an outlook for future work.

II. Problem Statement

The open source research project *ExplorViz* started in 2012 as part of a PhD thesis and is further developed and maintained until today. *ExplorViz* enables a live monitoring and visualization of large software landscapes [6], [7]. In particular, the tool offers two types of visualizations – a landscape-level and an application-level perspective. The first provides an overview of a monitored software landscape consisting of several servers, applications, and communication in-between. The second perspective visualizes a single application within the software landscape and reveals its underlying architecture, e.g., the package hierarchy in Java, and shows classes and related communication. The tool has the objective to aid the process of system and program comprehension for developers and operators. We successfully employed the software in several collaboration projects [8], [9] and experiments [10], [11]. The project is developed from the beginning on *Github* with a small set of core developers and many collaborators (more than 40 students) over the time. Several extensions have been implemented since the first version, which enhanced the tool's feature set. Unfortunately, this led to an unstructured architecture due to an unsuitable collaboration and integration process. In combination with technical debt and issues of our employed software framework and underlying architecture, we had to perform a technical and process-oriented modularization. Since 2012, several researchers, student assistants, and a total of 31 student theses as well as multiple projects contributed to *ExplorViz*. We initially chose the Java-based Google Web Toolkit (*GWT*) [12], which seemed to be a good fit in 2012, since Java is the most used language in our lectures. *GWT* provides different wrappers for Hypertext Markup Language (HTML) and compiles a set of Java classes to JavaScript (JS) to enable the execution of applications in web browsers. Employing *GWT* in our project resulted in a monolithic application (hereinafter referred to as *ExplorViz Legacy*), which introduced certain problems over the course of time.

A. Extensibility & Integrability

ExplorViz Legacy's concerns are divided in core logic (core), predefined software visualizations, and extensions. When *ExplorViz Legacy* was developed, students created new Git branches to implement their given task, e.g., a new feature. However, there was no extension mechanism that allowed the integration of features without rupturing the core's code base. Therefore, most students created different, but necessary features in varying classes for the same functionality. Furthermore, completely new technologies were utilized, which introduced new, sometimes even unnecessary (due to the lack of knowledge), dependencies. Eventually, most of the developed features could not be easily integrated into the master branch and thus remained isolated in their created feature branch.

B. Code Quality & Comprehensibility

After a short period of time, modern JS web frameworks became increasingly mature. Therefore, we started to use *GWT*'s JavaScript Native Interface (JSNI) to embed JS functionality in client-related Java methods. For example, this approach allowed us to introduce a more accessible JS-based rendering engine. Unfortunately, JSNI was overused and the result was a partitioning of the code base. Developers were now starting to write Java source code, only to access JS, HTML, and Cascading Style Sheets (CSS). This partitioning reduced the accessibility for new developers. Furthermore, the integration of modern JS libraries in order to improve the user experience in the frontend was problematic. Additionally, Google announced that JSNI would be removed with the upcoming release of Version 3, which required the migration of a majority of client-related code. Google also released a new web development programming language, named *DART*, which seemed to be the unofficial successor of *GWT*. Thus, we identified a potential risk, if we would perform a version update. Eventually, JSNI reduced our code quality. By code quality, we understand the maintainability of the source code, which includes the concepts of analyzability, changeability, and understandability [13]. Our remaining Java classes further suffered from ignoring some of the most common Java conventions and resulting bugs. Students of our university know and use supporting software for code quality, e.g., static analysis tools such as *Checkstyle* [14] or *PMD* [15]. However, we did not define a common code style supported by these tools in *ExplorViz Legacy*. Therefore, a vast amount of extensions required a lot of refactoring, especially when we planned to integrate a feature into the core.

C. Software Configuration & Delivery

In *ExplorViz Legacy*, integrated features were deeply coupled with the core and could not be easily taken out. Often, users did not need all features, but only a certain subset of the overall functionality. Therefore, we introduced new branches with different configurations for several use cases, e.g., a live demo. Afterwards, users could download resulting artifacts, but the maintenance of related branches was cumbersome. Summarized, the stated problems worsened the extensibility, maintainability, and comprehension for developers of our software. Therefore, we were in need of modularizing and modernizing *ExplorViz Legacy*.

III. *ExplorViz Legacy*

In order to understand the modularization process, we provide more detailed information about our old architecture in the following. The overall architecture and the employed

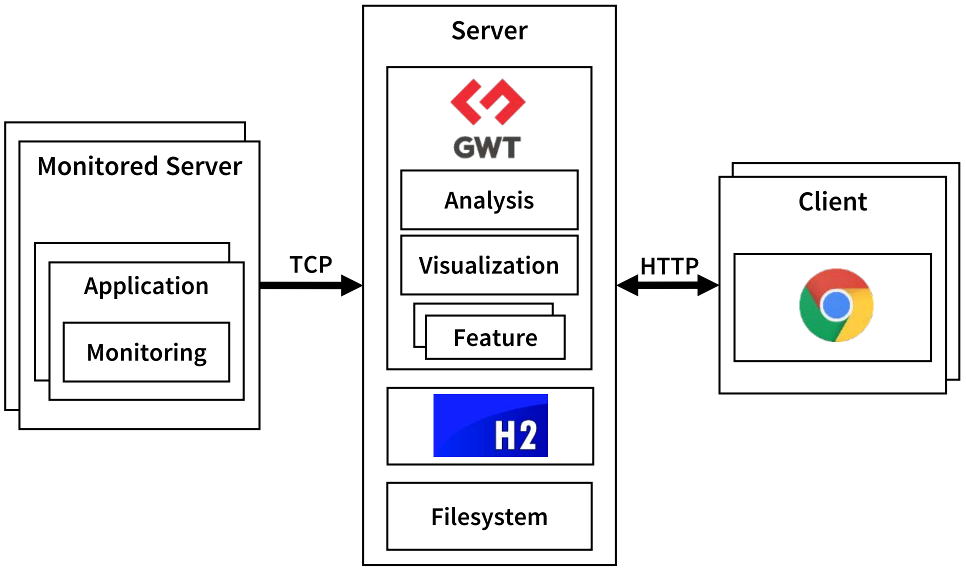

Figure 1: Architectural overview and software stack of the monolithic *ExplorViz Legacy*.

software stack of *ExplorViz Legacy* is shown in Figure 1. We are instrumenting applications, regardless whether they are native applications or deployed artifacts in an application server like *Apache Tomcat*. The instrumentation is realized by our monitoring component. The component employs in the case of Java *AspectJ*, an aspect-oriented programming extension for Java [16]. *AspectJ* allows us to intercept an application by bytecode-weaving. Thereby, we can gather necessary monitoring information for analysis and visualization purposes. Subsequently, this information is transported via Transmission Control Protocol (TCP) towards a server, which hosts our *GWT* application. This part represents the two major components of our architecture, namely *analysis* and *visualization*. The *analysis* component receives the monitoring information and reconstructs traces. These traces are stored in the file system and describe a software landscape consisting of monitored applications and communication in-between. Our user-management employs the *H2* database [17] to store related data. The software landscape *visualization* is provided via Hypertext Transfer Protocol (HTTP) and is accessible by clients with a web browser. *GWT* is an open source framework, which allows to develop JS front-end applications in Java. It facilitates the usage of Java code for server (backend) and client (frontend) logic in a single web project. Client-related components are compiled to respective JS code. The communication between frontend and backend is handled through asynchronous remote procedure calls (ARPC) based on HTTP. The usage of ARPC allows non-professional developers, in our case computer science students, to easily extend our existing open source research project. ARPC enables a simple exchange of Java objects between client and server. In *ExplorViz Legacy*, the advantages of *GWT* proved to be a drawback, because every change affects the whole project due to its single code base. New developed features were hard-wired into the software system. Thus, a feature could not be maintained, extended, or replaced by another component with reasonable effort. This situation was a leading motivation for us to look for an up-to-date framework replacement. We intended to take advantage of this situation and modularize our software system. The plan was to move from a monolithic to a distributed (web) application divided into separately maintainable and deployable backend and frontend components.

Our open source research project is publicly accessible since the beginning on *Github* and is licensed under the *Apache License, Version 2.0*. The development process facilitated the maintainability and extensibility of our software by means of so-called feature branches. Every code change, e.g., a new feature or bugfix, had to be implemented in a separated feature branch based on the master branch. This affected not only the core developers (researchers), but also student assistants, or students during a thesis or project. After performing a validation on the viability and quality of the newly written source code, the branch needed to be merged into the master project and thus permanently into the project. This fact often led to an intricate and time-consuming integration process, since all developers worked on a single code base. For that reason, we had to improve our development process to perform a modularization and technical modernization.

The previously mentioned drawbacks in *ExplorViz Legacy* were our initial trigger for a modularization and modernization. Additionally, recent experience reports in literature were published about successful applications of alternative technologies, e.g., Representational State Transfer (REST or RESTful) Application Programming Interfaces (API) [18], [19]. In the following, we describe our employed, iterative modularization and reengineering approach *CORAL*, which guided us through this process.

IV. The Modularization and Reengineering Approach *CORAL*

Our Collaborative Reengineering and Modularization Approach (*CORAL*) addresses problems regarding the modernization and modularization of open source research projects in technical and organizational aspects. This collaborative,

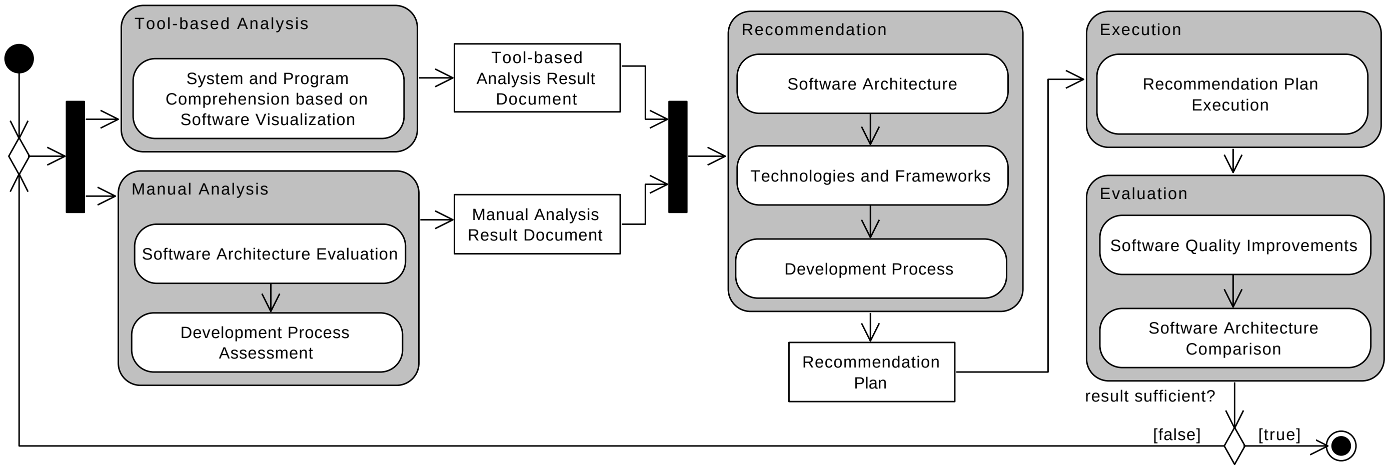
Tool-based Analysis
System and Program Comprehension based on Software Visualization
Manual Analysis
Software Architecture Evaluation
Development Process Assessment
Tool-based Analysis Result Document
Manual Analysis Result Document
Recommendation
Software Architecture
Technologies and Frameworks
Development Process
Recommendation Plan
Execution
Recommendation Plan Execution
Evaluation
Software Quality Improvements
Software Architecture Comparison
result sufficient?
[false]
[true]

Figure 2: UML activity diagram illustrating our iterative modularization and reengineering approach *CORAL*.

tool-employing approach supports developers and operators in modularizing and modernizing their software systems in an iterative manner. Basically, the approach consists of five, consecutive activities to support the modularization and reengineering of existing software projects and involved systems. Figure 2 gives an overview of the approach in form of an UML activity diagram. The five activities (colored in gray) are *Manual Analysis*, *Tool-based Analysis*, *Recommendation*, *Execution*, and *Evaluation*. In the following, the activities are briefly described.

A. *Manual Analysis*

An existing software project and involved systems, which are in need of modularization and modernization, have to be analyzed first (by the developers). Therefore, we need to take a look at the underlying architecture, employed technologies, and tools. This task includes a software architecture and modernization evaluation, in order to identify and reassess legacy source code, frameworks and utilized libraries, and execution environments. The software architecture evaluation task is divided into four parts – (i) a software architecture review, (ii) the application of the software architecture evaluation method *ATAM* [20], (iii) the identification of technical debt, and (iv) the examination of employed technologies and frameworks. For guidelines and approaches for evaluating software architectures, we refer to [21]–[23]. Additionally, the developers need to contribute their knowledge of known technical debt, existing documentation, and their current development process. For assessing and evaluating software development processes, we refer to [24], [25]. The results of this activity are summarized in form of a result document.

B. *Tool-based Analysis*

Afterwards, the system is analyzed with tools, which aid the modularization process by detecting (technical) flaws, possible shortcomings, and optimization potential. In detail, we focus on the aspect of understanding the software system. We address this aspect by employing the software visualization tool *ExplorViz* itself in order to aid the system and program comprehension process. We employ *ExplorViz* to achieve a better understanding of the software systems we want to modularize and modernize within our approach. *ExplorViz* was already successfully utilized for comprehension purposes in several scientific [8], [9] and industrial collaborations. By utilizing *ExplorViz* for the program comprehension process, we take advantage of software visualizations instead of software artifacts like source code or documentation. Thus, we can enhance our previously obtained knowledge about the software systems from discussions and interviews with the software developers. Finally, we document our findings in form of a result document.

C. *Recommendation*

In this activity, we take a look into the analysis result documents of the *Manual Analysis* and *Tool-based Analysis* activities, and design a recommendation plan in collaboration with the developers. The recommendation plan is based on the results and examines possible (target) architectures, technologies, and frameworks. Thereby, we also take the employed development process into account. The purpose is to facilitate synergy effects between the software system and the corresponding development process. In the best case, we achieve a collaborative development process, which supports the planned modularization and modernization from the beginning.

D. *Execution*

After discussing the presented options leading towards a recommendation plan in the last activity, we need to prepare the execution of it. More precisely, we work out a proof-of-concept implementation of the recommendation plan first. Thus, we can verify the necessary technical adaptions in general and are able to perform the reengineering and modularization process afterwards on a solid basis.

E. *Evaluation*

Once we executed our recommendation plan, we need to evaluate its impact on the software system. Therefore, we focus on comparing the software quality based on metrics provided by software quality tools on one hand and the software architecture through visual comparison on the other hand. Typically, the results of the evaluation are not sufficient after only one execution. Thus, it is likely, that the overall approach needs to be conducted multiple times in order to achieve an acceptable state.

V. First Iteration: Modularization Process and Architecture of *ExplorViz*

Within *ExplorViz Legacy*, we applied the above mentioned process, which guided us through our modularization process from performing a first requirement analysis and defining goals towards our actual state. Summarized, we performed multiple iterations of the process until we reached an entire, maintainable, and especially extensible microservice architecture. In the following, the first iteration of the process is described.

A. Requirement Analysis and Goals

We no longer perceived advantages of preferring *GWT* over other web frameworks. During the modularization planning phase, we started with a requirement analysis for our modernized software system and identified technical and development process related impediments in the project. We kept in mind that our focus was to provide a collaborative development process, which encourages developers to participate in our research project [26]. Furthermore, developers, especially inexperienced ones, tend to have potential biases during the development of software, e.g., they make decisions on their existing knowledge instead of exploring unknown solutions [27].

As a result, we intended to provide plug-in mechanisms for the extension of the backend and frontend with well-defined interfaces. We intended to encourage developers to try out new libraries and technologies, without rupturing existing code. According to [28], the organization of a software system implementation is not an adequate representation of a system's architecture. Thus, architectural changes towards the implementation of a software system have to be documented before or at least shortly after the realization. If this aspect is not addressed, the architecture model has a least to be updated based on the implementation in a timely manner. Thus, we took this into account in order to enhance our development process. Architectural decay in long-living software systems is also an important aspect. Over time, architectural smells manifest themselves into a system's implementation, whether they were introduced into the system from the beginning or later during development [29]. For the modularization process of our software system it was necessary to look for such smells to eliminate them in the new system. In the end, we identified the following goals for our modularization and modernization process:

- The project needs to be stripped down to it's core, anything else is a form of extension.
- We need to focus on the main purpose of our project – the visualization of software landscapes and architectures. Thus, we need to look for a monitoring alternative.
- The backend and frontend should be separately deployable and technologically independent. The latter goal allows us to replace them with little effort. Additionally, they store their own data and use no centralized storage or database.
- Scaffolds or dummy-projects are provided for the development of extensions.
- We stick to the encapsulation principle and provide well-defined interfaces.
- The overall development process needs to be enhanced, e.g, by using Continuous Integration (CI) and quality assurance (QA), like code quality checks.

In general, there exist many drivers and barriers for microservice adoption [30]. Typical barriers and challenges are the required additional governance of distributed, networked systems and the decentralized persistence of data. After we applied the two activities *Manual Analysis* and *Tools-based Analysis* within our iterative *CORAL* approach, we agreed within the *Recommendation* activity to build our recommendation plan upon an architecture based on microservices. This architectural style offers the ability to divide monolithic applications into small, lightweight, and independent services, which are also separately deployable [4], [31]–[33]. However, the obtained benefits of a microservice architecture can bring along some drawbacks, such as increased overall complexity and data consistency issues [34]. Adopting the above mentioned goals lead us finally to the microservice-based architecture shown in Figure 3.

B. Extensibility & Integrability

In a first step, we modularized our *GWT* project into two separated projects, i.e., backend and frontend, which are now two self-contained microservices. Thus, they can be developed technologically independent and deployed on different server nodes. In detail, we employ distinct technology stacks with independent data storage. This allows us to replace the microservices, as long as we take our specified APIs into account. We tried to evaluate how we can facilitate the development for our main collaborators, i.e., our students. Therefore, our selection of technologies was driven by the students' education at the Kiel University. The backend is implemented as a Java-based web service based on *Jersey* [35], which provides a RESTful API via HTTP for clients. We chose Jersey, because of its JAX-RS compliance. In our opinion, Jersey is a mature framework and due to its HTTP roots it is easy to understand for developers, especially collaborators such as students. *Jersey* implements the Servlet 3.0 specification, which offers javax.servlet.annotations to define servlet declarations and mappings. We assume that the usage of the Servlet 3.0 specification eases the development process in the backend, especially for students. Furthermore, we replaced our custom-made monitoring component by the monitoring framework *Kieker* [36]. This framework provides an extensible approach for monitoring and analyzing the runtime behavior of distributed software systems. Monitored information is sent via TCP to our backend, which employs the filesystem and *H2* database for storage. *Kieker* employs a similar monitoring data structure, which fits our replacement requirements perfectly. The frontend uses the JS framework *Ember.js*, which enables us to offer visualizations of software landscapes to clients with a web browser [37]. *Ember.js* was chosen, since its core idea of using addons to modularize the application, i.e., the frontend of *ExplorViz* is a good practice in general. Furthermore, the software ecosystem of *Ember.js* with community-driven addons is tremendous and their developer team frequently updates the framework with new features. Since *Ember.js* is based on the *model-view-viewmodel* architectural pattern, developers do not need to manually access the Document Object Model and thus need to write less source code. *Ember.js* uses *Node.js* as execution environment and emphasizes the use of components in web sites, i.e., self-contained, reusable, and exchangeable user interface fragments [38]. We build upon these components to encapsulate distinct visualization modes, especially for extensions. Communication, like a request of a software landscape from the backend, is abstracted by so-called *Ember.js* adapters. These adapters make it easy to request or send data by using the *convention-over-configuration* pattern. The introduced microservices, namely backend and frontend, represent the core of *ExplorViz*. As for future extensions, we implemented well-defined extension interfaces for both microservices, that allow their integration into the core.

C. Code Quality & Comprehensibility

New project developers, e.g., students, do not have to understand the complete project from the beginning. They can now extend the core by implementing new mechanics on the basis of a plug-in extension. Extensions can access the core functionality only by a well-defined read-only API, which is implemented by the backend, respectively frontend. This high level of encapsulation and modularization allows us to improve the project, while not breaking extension support. Additionally, we do no longer have a conglomeration between backend and frontend source code, especially the mix of Java and JS, in single components. This eased the development process and thus reduced the number of bugs, which previously occurred in *ExplorViz Legacy*. Another simplification was the use of *json:api* [39] as data exchange format specification between backend and frontend, which introduced a well-defined JavaScript Object Notation (JSON) format with attributes and relations for data objects. This minimizes the amount of data and round trips needed when making API calls. Due to its well-defined structure and relationship handling, developers are greatly supported when exchanging data.

D. Software Configuration & Delivery

One of our goals was the ability to easily replace the microservices. We fulfill this task by employing frameworks, which are exchangeable with respect to their language domain, i.e., Java and JS. We anticipate that substituting these frameworks could be done with reasonable effort, if necessary. Furthermore, we offer pre-configured artifacts of our software for several use cases by employing *Docker* images. Thus, we are able to provide containers for the backend and frontend or special purposes, e.g., a fully functional live demo. Additionally, we implemented the capability to plug-in developed extensions in the backend, by providing a package-scanning mechanism. The mechanism scans a specific folder for compiled extensions and integrates them at runtime.

VI. Proof-of-Concept Implementation

In order to execute and afterwards evaluate the recommendation plan we designed before, we realized a proof-of-concept implementation and split our project as planned into two separate projects – a backend project based on *Jersey*, and a frontend project employing the JS framework *Ember.js*. Both frameworks have a large and active community and offer sufficient documentation, which is important for new developers. As shown in Figure 3, we strive for an easily maintainable, extensible, and plug-in-oriented microservice architecture. Since the end the first iteration of our modularization and modernization process in early 2018, we were able to successfully develop several extensions both for the backend and the frontend. Four of them are described in the following.

A. Application Discovery

Although we employ the monitoring framework *Kieker*, it lacks a user-friendly, automated setup configuration due to its framework characteristics. Thus, users of *ExplorViz* experienced problems with instrumenting their applications for monitoring. In [40], we reported on our application discovery and monitoring management system to circumvent this drawback. The key concept is to utilize a software agent that simplifies the discovery of running applications within operating systems. An example visualization of the extension's user-interface is shown in Figure 4. The figure shows three discovered applications on a monitored server. Furthermore, this extension properly configures and manages the monitoring framework *Kieker*. More precisely, the extension is divided in a frontend extension, providing a configuration interface for the user, and a backend extension, which applies this configuration to the respective software agent lying on a software system. Then, the software agent is able to apply the chosen configuration towards *Kieker* for the application monitoring.

Finally, we were able to conduct a first pilot study to evaluate the usability of our approach with respect to an easy-to-use application monitoring. The improvement regarding the usability of the monitoring procedure of this extension was a great success. Thus, we recommend this extension for every user of *ExplorViz*.

B. Virtual Reality Support

An established way to understand the complexity of a software system is to employ visualizations of software landscapes. However, with the help of visualization alone, exploring an unknown software system is still a potentially challenging and time-consuming task. In the past years, Virtual Reality (VR) techniques emerged at the consumer market. Starting with the Oculus Rift DK1 head-mounted display (HMD), which was available at the end of 2013, the VR devices constituted a major step towards the consumer market. Based on this development, modern VR approaches became affordable and available for various research purposes. A similar development can be observed in the field of gesture-based interfaces, when Microsoft released their Kinect sensor in 2010 [41]. A combination of both techniques offers new visualization and interaction capabilities for newly created software, but can also improve reverse engineering tasks of existing software by means of immersive user experience.

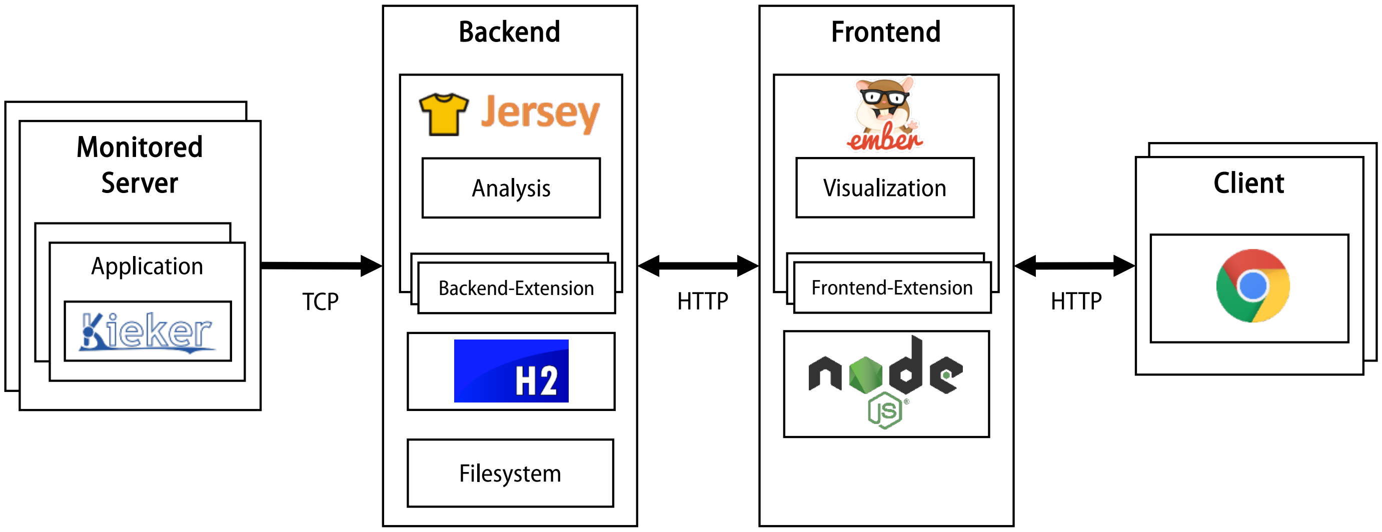

Figure 3: Architectural overview and software stack of the modularized *ExplorViz* (after the first iteration).

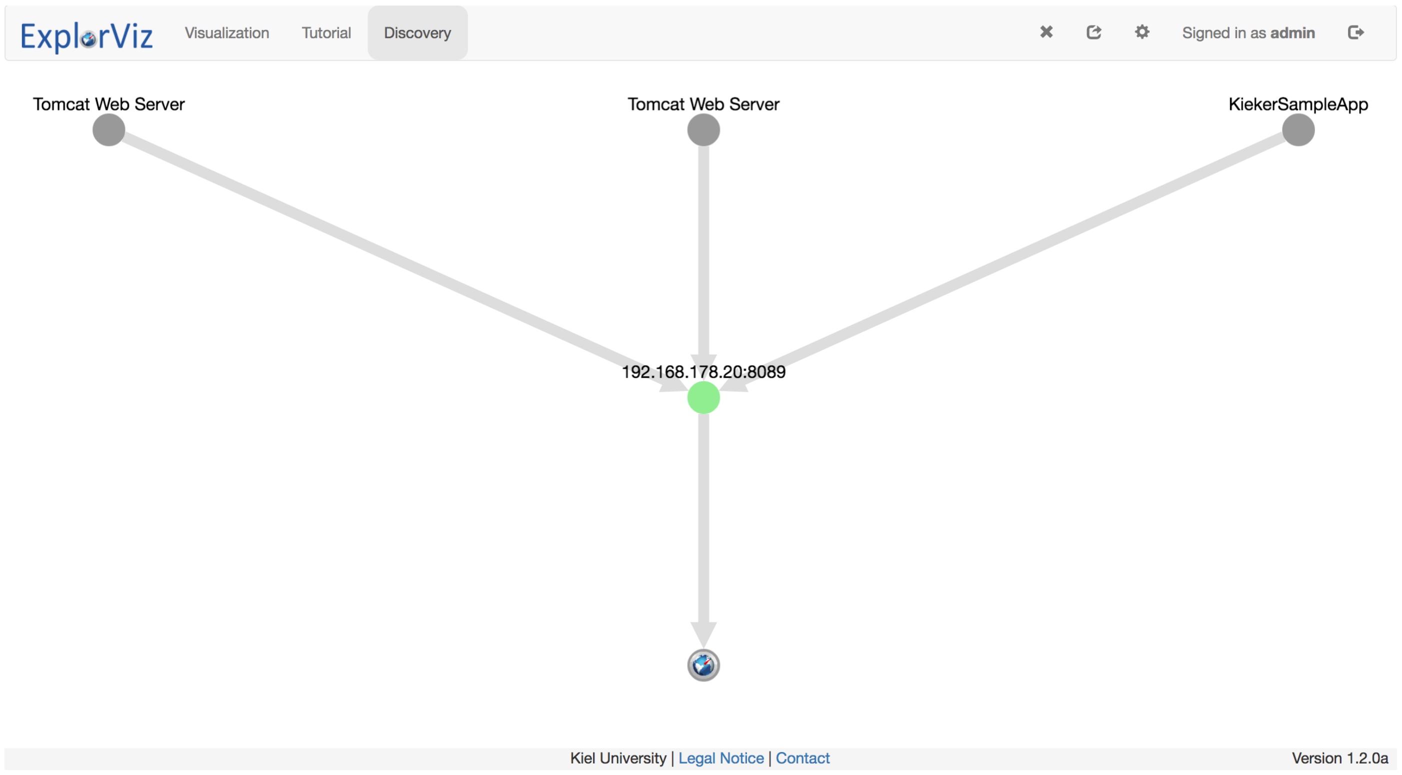

Figure 4: Screenshot of the application discovery extension of *ExplorViz*.

Based on an in-depth 3D visualization and a more natural interaction, compared to a traditional 2D screen and input devices like mouse and keyboard, the user gets a more immersive experience, which benefits the comprehension process [42]. VR can offer an advantage in comparison to existing developer environments to enable new creative opportunities and potentially result in higher productivity, lower learning curves, and increased user satisfaction [43]. For this extension, five students followed a new approach using VR for exploring software landscapes collaboratively based on our previous work [44]. They employed severals HMDs (HTC Vive, HTC Vive Pro, and Oculus Rift) to allow a collaborative exploration and comprehension of software in VR. A screenshot of the VR extension featuring the application-perspective and visualized VR controllers is shown in Figure 5. The collaborative VR approach builds upon our microservice architecture and employs WebSocket connections to exchange data to achieve modular extensibility and high performance for this real-time

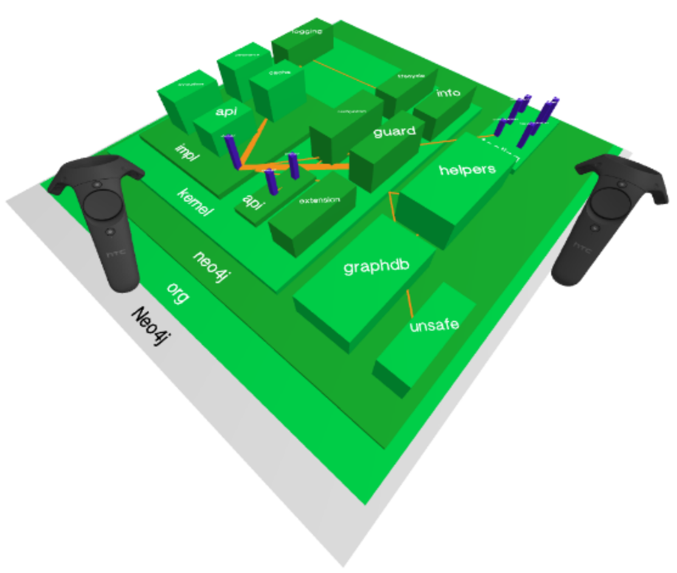

Figure 5: Screenshot of the VR extension of *ExplorViz* showing the application perspective and visualized VR controllers.

multi-user environment. As a proof of concept, they conducted a first usability evaluation with 22 subjects. The results of this evaluation revealed a good usability and thus constituted a valuable extension to *ExplorViz*. Recently, we performed a second evaluation focusing on the applicability of the approach for system and program comprehension tasks in teams. With 24 subjects, grouped into physically separated teams of two persons, they solved comprehension tasks collaboratively. First results indicated an efficient usage of the approach, which could offer an alternative to traditional 2D displays and interaction devices.

C. Architecture Conformance Checking

Software landscapes evolve over the time, and consequently, architecture erosion occurs. This erosion causes high maintenance and operation costs, thus performing architecture conformance checking (ACC) is an important task. ACC allows faster functionality changes and eases the adaptation to new challenges or requirements. Additionally, software architects can use ACC to verify a developed version against a previous modeled version. This can be used to check whether the current architecture complies with the specified architecture and allows to reveal constraint violations. An example architecture conformance visualization of a monitored software landscape against a modeled one is shown in Figure 6. The visualization illustrates missing or modified (colored in red), and additional (colored in blue) nodes and applications and related communication in-between for a software landscape. In this extension, a student developed an approach to perform ACC between a modeled software landscape consisting of applications using an editor and a monitored software landscape. This allows us to perform a visual comparison between both versions on an architectural level. In order to evaluate the extension, the student conducted a usability study with five participants, applying the model editor for a desired software landscape and performing ACC of a modeled software landscape against a monitored one. The results indicated a good user experience of the approach, although the usability of the editor could be improved.

D. Visualizing Architecture Comparison

Identifying architectural changes between two visualizations of a complex software application is a challenging task, which can be supported by appropriate tooling. Although *ExplorViz* visualizes the behavior and thus the runtime architecture of a software system, it is not possible to compare two versions. In this extension one student developed an approach to perform a visual software architecture comparison of two monitored applications, e.g., indicating removed or changed components or classes. This facilitates a developer to see at a glance which parts of the architecture have been added, deleted, modified, or remained unchanged between the two versions. Finally, an evaluation based on a qualitative usability study with an industrial partner was conducted. Five professional software engineers participated in the study and solved comparison tasks based on two different versions of their own developed software. The evaluation showed that the extension is applicable for solving architecture comprehension tasks with different versions within *ExplorViz*.

VII. Second Iteration: Restructured Architecture and new Process

As the evaluations at the end of the first iteration revealed some drawbacks, we decided to perform a second iteration of our modularization and modernization approach. After evaluating the first iteration we identified, among others, four major drawbacks, which are presented in the following.

- Extensibility & Integrability: Higher services had to perform several HTTP requests to obtain necessary information from variety of services.
- Code Quality & Comprehensibility: The coding quality was on a low level due to the lack of employed QA tools and rules.
- Software Configuration & Delivery: We needed to provide compiled Java files of all available backend extensions.
- Software Architecture Erosion & Accessibility: The configuration of the monitoring was still too difficult.

Our modularization approach started by dividing the old monolith into separated frontend and backend projects [26]. Since then, we further decomposed our backend into several microservices to address the problems stated in Section II. The resulting, restructured architecture is illustrated in Figure 7 and the new collaborative development process is described below. As reported in Section VI, the new architecture already improved the collaboration with new developers who realized new features as modular extensions.

A. Extensibility & Integrability

Frontend extensions are based on *Ember.js*'s addon mechanism. This approach works quite well for us as shown in Section VI. The backend, however, used the package scanning feature of *Jersey* to include extensions. The result of this

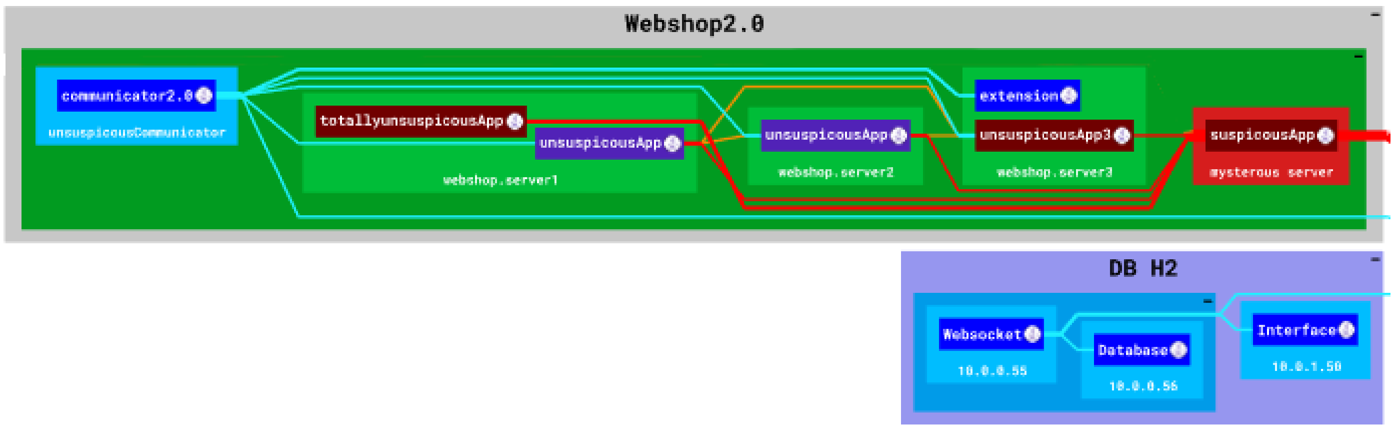

Figure 6: Screenshot of the architecture conformance checking extension of *ExplorViz*.

procedure was again an unhandy configuration of a monolithic application with high coupling of its modules. Therefore, we once again restructured the approach for our backend plug-in extensions. The extensions are now decoupled and represent separated microservices. As a result, each extension is responsible for its own data persistence and error handling. Due to the decomposition of the backend, we are left with multiple Uniform Resource Identifiers (URI). Furthermore, new extensions will introduce additional endpoints, therefore more URIs again. To simplify the data exchange handling based on those endpoints, we employ a common approach for microservice-based backends. The frontend communicates with an API gateway instead of several single servers, thus only a single base Uniform Resource Locator (URL) with well-defined, multiple URIs. This gateway, a *NGINX* reverse proxy [45], passes requests based on their URI to the respective proxied microservices, e.g., the *landscape-service*. Furthermore, the gateway acts as a single interface for extensions and offers additional features like caching and load balancing. Extension developers, who require a backend component, extend the gateway's configuration file, such that their frontend extension can access their complement. Some extensions must read data from different services. In the past, we used HTTP requests to periodically obtain this data. Each request was processed by the providing service, therefore introducing unnecessary load. The inter-service communication is now realized with the help of *Apache Kafka* [46]. *Kafka* is a distributed streaming platform with fault-tolerance for loosely coupled systems. We use *Kafka* for events that might be interesting for upcoming microservices. For example, the *landscape-service* consumes traces from the respective *Kafka* topic and produces a new landscape every tenth second for another topic. Microservices can consume the topic, obtain, and process the data in their custom way. As a result, the producing service does not have to process unnecessary HTTP requests, but simply fires its data and forgets it. Simple Create Read Update Delete (CRUD) operations on resources, e.g., users and their management, are provided by means of RESTful APIs by the respective microservices. The decomposition into several independent microservices and the new inter-service communication approach both facilitate low coupling in our system.

B. Code Quality & Comprehensibility

The improvements for code quality and accessibility, which were introduced in the first iteration of our modularization approach, showed a perceptible impact on contributor's work. For example, recurring students approved the easier access to *ExplorViz* and especially the obligatory exchange format *json:api*. However, we still lacked a common code style in terms of conventions and best practices. To achieve this and therefore facilitate maintainability, we defined compulsory rule sets for the quality assurance tools *Checkstyle* and *PMD*. In addition with *SpotBugs* [47], we impose their usage on contributors for Java code. For JS, we employ *ESLint* [48], i.e., a static analysis linter, with an *Ember.js* community-driven rule set. The latter contains best practices for *Ember.js* applications and rules to prevent programming flaws. In the future, we are going to enhance this rule set with our custom guidelines. Another aspect are CI tools. CI systems and tools are used to automate the compilation, building, and testing of software (systems). Software projects that employ CI, release twice as often, accept pull requests faster, and have developers who are less worried about breaking the build, compared to projects that do not use CI [49]. Therefore, employing CI tools is a good method to improve our development process even more. Consequently, we integrated the previously mentioned tools into our continuous integration pipeline configured in *TravisCI* [50]. More precisely, we employ *TravisCI* for *ExplorViz*'s core and any extension to build, test, and examine the code. Integrating the quality assurance tools allows us to define thresholds within the pipeline. If a threshold regarding quality assurance problems is exceeded, the respective *TravisCI* build will fail and the contributor is notified by mail. A similar build is started for each pull request that we receive on *Github* for the now protected master branch. Therefore, contributors are forced to create a new branch or fork *ExplorViz* to implement their enhancement or bug fix and eventually submit a pull request.

C. Software Configuration & Delivery

One major problem of *ExplorViz Legacy* was the necessary provision of software configurations for different use cases. The first iteration of modularization did not entirely solve this problem. The backend introduced a first approach for an

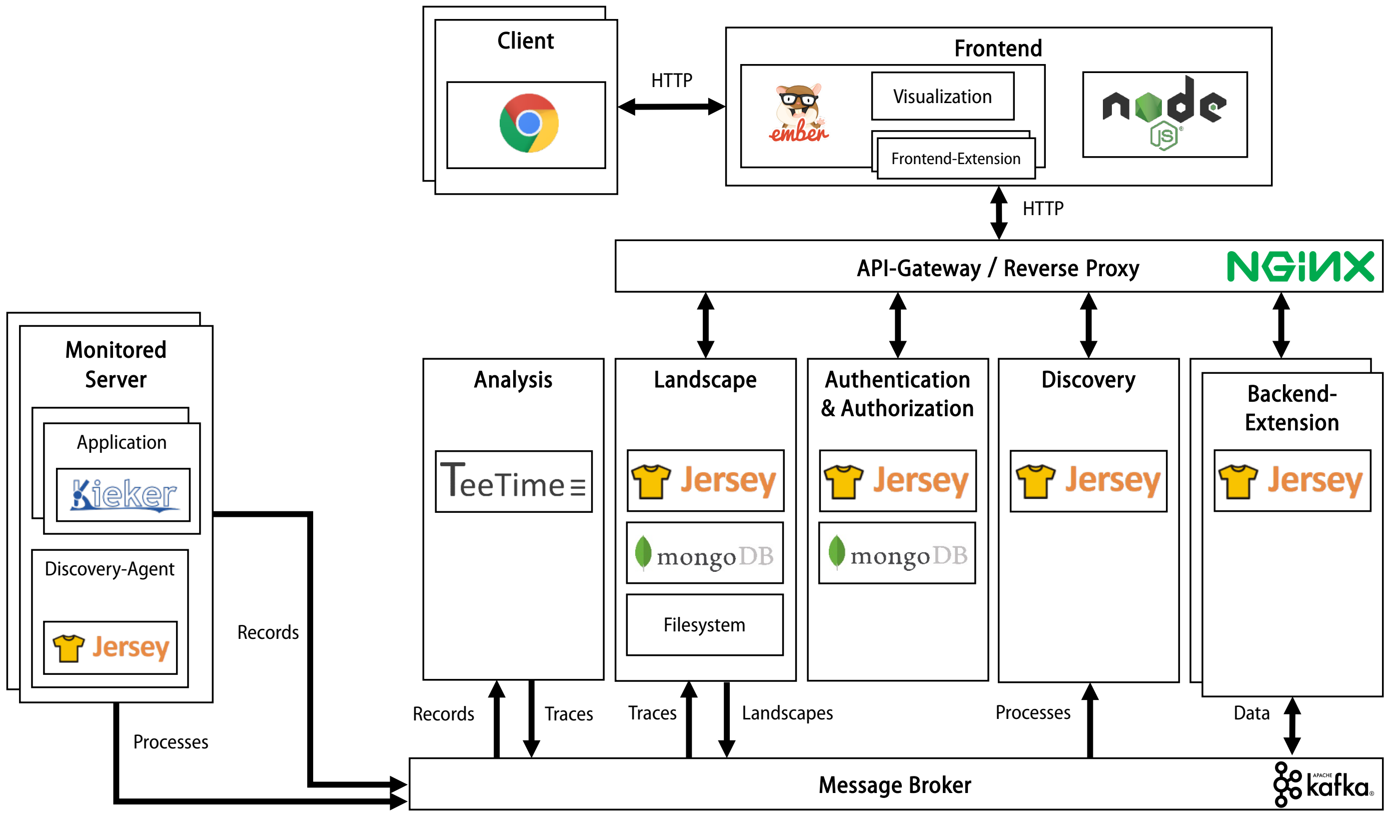

Figure 7: Architectural overview and software stack of *ExplorViz* (after the second iteration).

integration of extensions, but their delivery was cumbersome. Due to the tight coupling at source code level we had to provide the compiled Java files of all extensions for download. Users had to copy these files to a specific folder in their already deployed *ExplorViz* backend. Therefore, configuration alterations were troublesome. With the architecture depicted in Figure 7 we can now provide a jar file for each service with an embedded web server. This modern approach for Java web applications facilitates the delivery and configuration of *ExplorViz*'s backend components. In the future, we are going to ship ready-to-use *Docker* images for each part of our software. The build of these images will be integrated into our CI pipeline. Users are then able to employ docker-compose files to achieve their custom *ExplorViz* configuration or use a provided docker-compose file that fits their needs. As a result, we can provide an alternative, easy to use, and exchangeable configuration approach that essentially only requires a single command line instruction. The frontend requires another approach, since (to the best of our knowledge) it is not possible to install an *Ember.js* addon inside of a deployed *Ember.js* application. We are currently developing a build service for users that ships ready-to-use, pre-built configurations of our frontend. Users can then download and deploy these pre-built packages. Alternatively, these configurations will also be usable as *Docker* containers.

D. Software Architecture Erosion & Accessibility

One of our initial problems was the partitioning of our code base and the resulting software architecture erosion. We think that both employed frameworks, *Ember.js* and *Jersey*, matter when it comes to this problem. *Ember.js* is well documented and there are many examples on how to solve a problem with the framework. Due to its JS nature, we can easily introduce and use modern features in web development. Furthermore, *Ember.js* introduces recognizable and reusable structures which facilitate the development. For the *Jersey* backend, we again provide a sample project that contributors can use for a start. The project is runnable and shows how to use *Kafka* and the HTTP client for different needs. *ExplorViz* uses the monitoring framework *Kieker* to obtain monitoring data. These so called Records are then processed by the analysis component of our software. The setup of *Kieker* is extensive, but also quite complex for untrained users. Since we are dealing with many students, we were in need of a solution to circumvent this drawback. We developed an external component with a frontend and backend extension that simplifies the monitoring setup for users. The so called discovery agent searches for running Java processes in the encompassing operating system and sends its data to the related discovery backend extension. The frontend discovery extension visualizes the gathered data and provides Graphical User Interface (GUI) forms for users to start and stop the monitoring of found processes. Ultimately, the resulting discovery mode was successful in internal tests and we integrated it as a core feature.

VIII. Third Iteration: Achieving an entire Microservice Architecture

The second iteration of our modernization process introduced multiple microservices for different backend logic. For example, each backend extension was build as a separate source code project and deployed as Java jar file. This introduced advantages, among others, for the configuration of *ExplorViz* as described in Section VII. Since then, we further refined our microservice decomposition. The current architecture, after performing a third iteration of our modularization approach, is illustrated in Figure 8. Additionally, we revised the ubiquitous problems revealed within the evaluation, as we did in the previous iterations. Thus, we identified, among others, three major drawbacks, which are presented in the following.

- Extensibility & Integrability: Implementing extensions against specific backend or frontend versions was difficult.
- Code Quality & Comprehensibility: Collaborators like students, did not have enough documentation to efficiently contribute to our project.
- Software Configuration & Delivery: The testing and release management was still cumbersome due to a vast number of artifacts.

A. Extensibility & Integrability

Both previous iterations shared the problem that collaborators had to implement their feature or extension against the latest version of *ExplorViz*. To circumvent this drawback, we now push the backend build artifacts of the *TravisCI* build pipeline as snapshots to *Sonatype* [51], i.e., an online maven repository for unsigned artifacts. Furthermore, we use *Github* releases to version *ExplorViz*. These releases follow a documented release management process. As a result, release descriptions and names share a common theme. In general, *Github* releases use Git tags to reference the specific Git commit that represents the release. We use these resulting Git tags for versioning. The tags are picked up by our CI pipeline and are used to name the *Sonatype* snapshots. As a result, contributors can now select specific (intermediate) versions to implement against.

After employing the second iteration of our modernization for some time with different configurations, we observed performance issues regarding the *landscape-service*. This service continuously built our hierarchical landscape model, provided the latest snapshot of the model via a HTTP API, and returned previous snapshots upon incoming HTTP requests. We identified that we could decompose these functionalities into separated microservices to distribute the load on one hand and gain a better performance on the other hand. The decoupling of the *landscape-service* can be seen in Figure 8. Frontend extensions now register at the *broadcast-service* to receive server-sent events (SSE), which contain the latest landscape model snapshot. Furthermore, specific snapshots can be requested at the *history-service*. This microservice is responsible for storing landscape model snapshots.

B. Code Quality & Comprehensibility

Introducing static analysis tools to our CI pipeline showed improvements of *ExplorViz*'s code style. The automatic CI build for *Github* pull requests highlights flaws and allows us to impose refactoring before merging the code. This is also used for collaborators' extensions. Now, the remaining part to improve the overall code quality was testing the source code and the integration of components. We observed that collaborators had less problems with testing frontend extensions than with testing the related backend project. We think that is due to the *Ember.js* documentation and the huge number of already existing open source projects, which already show how one can comprehensively test *Ember.js* projects. Therefore, we wrote sample unit, integration, and API tests for our microservices, which students can use as foundation to test their own written code. By choosing these three categories of tests, we now cover testing at source code and API level. All these tests are automatically executed as part of our CI pipeline. Furthermore, when a tests requires other running services, e.g., the reverse proxy, these services are (if necessary) build and executed by means of a *Docker* container.

To ease the development for collaborators, we wrote supplemental guides on best practices, design ideas, and specifications. These can be found in our public *Github* documentation wiki [52]. Furthermore, our CI pipeline now automatically builds the latest API documentation (*JavaDoc* for the backend and *YUIDoc* for the frontend). The resulting websites are deployed by means of *Github* pages, i.e., public websites based on the content of Git repositories. We additionally employ *Swagger* [53], an interactive API development editor and UI, to document our HTTP APIs. The tool is automatically started when a microservice is started in development mode.

C. Software Configuration & Delivery

ExplorViz enables users and developers to use extensions on demand by providing the build artifacts for every (release) version. We now facilitate *ExplorViz*' configuration with the help of *Docker* images. After pushing the build artifacts to *Sonatype* in the CI pipeline, we subsequently build a *Docker* image for each service and push it to *Docker Hub*. Therefore, users and collaborators can use the publicly hosted *Docker* images to easily create their custom deployment environment with *Docker*.

We build upon this process and now provide ready-to-use docker-compose files for release versions of *ExplorViz*. These configurations allow users to start the core features of *ExplorViz* with only a single command. This approach is also used in the development phase. Since *ExplorViz* requires auxiliary software, i.e., database management systems, *Apache Kafka*, and the reverse proxy *NGINX*, we now provide a docker-compose file to start the mandatory, already configured software stack for development. As a result, collaborators do not need to read different instructions on how to start specific software, but only need to start a set of *Docker* containers with the help of the docker-compose file.

Figure 8 shows that we replaced our employed reverse proxy *NGINX* with *Traefik* [54]. The reverse proxy *NGINX* uses a static configuration file to define its routing. As a result, *ExplorViz* users needed to update this configuration or use a

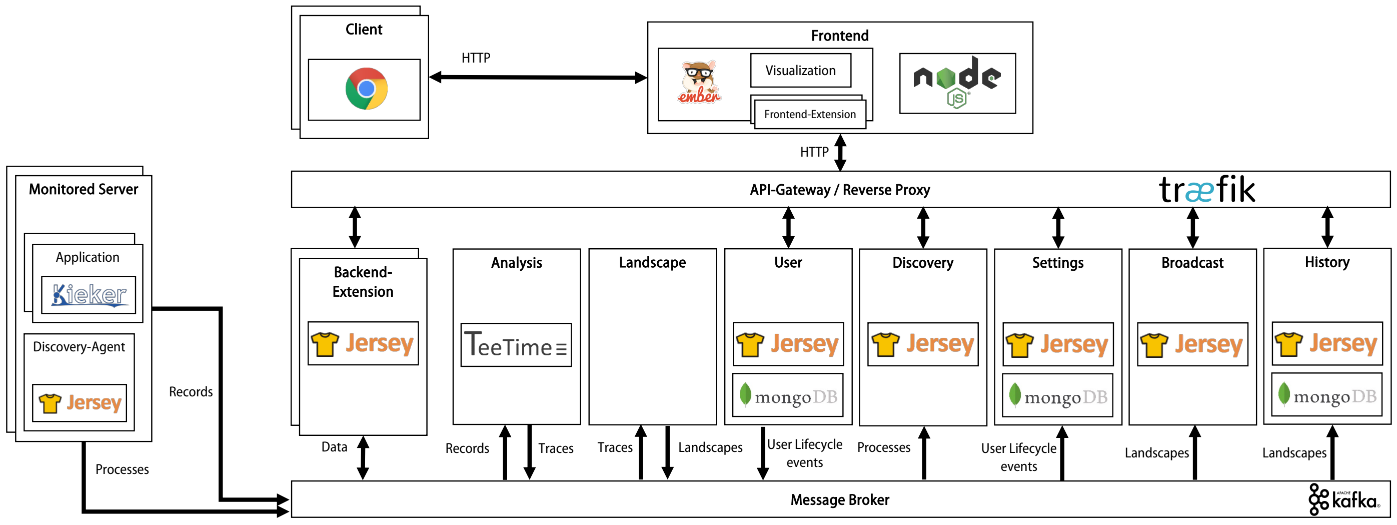

Figure 8: Current architectural overview and software stack of *ExplorViz* (after the third iteration).

provided version to enable an installed or developed extension. This was quite cumbersome and potentially deterred users to try out extensions. With *Traefik* we can now use labels, i.e., metadata for *Docker* objects, to define the routing at docker-compose level. Therefore, the routing of the reverse proxy can be easily extended or changed.

IX. Related Work

In the area of software engineering, there are many papers that perform a software modernization in other contexts. Thus, we restrict our related work to approaches, which focus on the modernization of monolithic applications towards a microservice architecture. Compared to frequently performed software modernizations, we did not reconstruct the underlying software architecture, since it was not our goal to keep the obsolete monolithic architecture provided by *GWT*. Furthermore, we did not need to apply multiple refactoring iterations to modernize our software system. Instead, we successfully performed three iterations of our modularization and modernization process *CORAL* in order to continuously improve our software architecture and collaborative development process.

Villamizar et al. [55] evaluate monolithic and microservice architectures regarding the development and cloud deployment of enterprise applications. Their approach addresses similar elements to our modernization process. They employed modern technologies for separating microservices, e.g., Java in the backend and JS in the frontend, like we did. Contrary to their results, we did not face any of the mentioned problems during the migration, like failures or timeouts. In [56] an approach regarding the challenges of the modernization of legacy J2EE applications was presented. They employ static code analysis to reconstruct architectural diagrams, which then can be used as a starting point during a modernization process. In contrast to our approach there was no need to reconstruct the software architecture, because we wanted to modernize it from the beginning due to previously mentioned drawbacks. Thus, we split our application based on our knowledge into several microservices and developed a communication concept based on a message broker. Carrasco et. al [34] present a survey of architectural smells during the modernization towards a microservice architecture. They identified nine common pitfalls in terms of bad smells and provided potential solutions for them. *ExplorViz Legacy* was also covered by this survey and categorized by the “Single DevOps toolchain” pitfall. This pitfall concerns the usage of a single toolchain for all microservices. Fortunately, we addressed this pitfall since their observation during their survey by employing independent toolchains by means of pipelines within our continuous integration system for the backend and frontend microservices.

Knoche and Hasselbring [31] present a migration process to decompose an existing software system into several microservices. Additionally, they report from their gained experiences towards applying their presented approach in a legacy modernization project. Although their modernization drivers and goals are similar to our procedure, their approach features a more abstract point of view on the modernization process. Furthermore, they focus on programming language modernization and transaction systems. In [4], the authors present an industrial case study concerning the evolution of a long-living software system, namely a large e-commerce application. The addressed monolithic legacy software system was replaced by a microservice-based system. Compared to our approach, this system was completely rebuilt without retaining code from the (commercial) legacy software system. Our focus is to facilitate the collaborative development of open source software and also addresses the development process. We successfully developed our pipeline towards CI for all microservices mentioned in Section VII to minimize the release cycles and offer development snapshots.

A different approach to perform a modernization of a monolithic application is presented in [57]. They employed a Domain-Driven Design (DDD) based approach to decomposition their software system into services. Afterwards, they integrated the services with an Enterprise Bus and orchestrated the services on the basis of Docker Compose and Swarm. In contrast to their approach, we did not perform a decomposition of our monolithic application based on DDD. Instead, we performed a decomposition based on backend and frontend logic within our first iteration and refined it later. Additionally, we employ Docker images for the deployment of *ExplorViz* and do not use Docker swarm. Chen et. al [58] present a top-down based dataflow-driven approach as an alternative decomposition method. More precisely, they developed a dataflow-driven decomposition algorithm, which operates on the basis of a constructed dataflow diagram modeling the business logic of the software system. In the next step, the dataflow diagram is compacted based on similar operations with the same type of output data. Finally, microservice candidates are identified and extracted. In comparison, our approach does not facilitate the usage of an algorithm which aids the decomposition process and identifies microservice candidates. In detail, we propose a collaborative-oriented, iterative process, which contains multiple steps and also addresses the involved development process.

X. Conclusion

In the following, we conclude our paper and present a summary, depict lessons learned, and give an outlook for future work.

A. Summary

In this paper, we reported on our modularization and modernization process of the open source research software *ExplorViz*, moving from a monolithic architecture towards a microservice architecture with the primary goal to ease the collaborative development, especially with students. We described technical and development process related drawbacks of our initial project state until 2016 in *ExplorViz Legacy* and illustrated our modularization process and architecture. The process included not only a decomposition of our web-based application into several components, but also a technical modernization of applied frameworks and libraries. Driven by the goal to easily extend our project in the future and facilitate a contribution by inexperienced collaborators, we offer a plug-in extension mechanism for our core project, both for backend and frontend. On the basis of *ExplorViz Legacy*, we employed our iterative, collaborative modularization and reengineering process *CORAL* as a guidance through our modularization and performed three successful iterations to *ExplorViz Legacy* until we reached a sufficient state.

After our first iteration, we realized our modularization process and architecture in terms of a proof-of-concept implementation and evaluated it afterwards by the development of several extensions of *ExplorViz*. Each of these extensions was developed by students and evaluated afterwards, in each case by at least a usability study. The results showed an overall good usability of each extension. In the case of our developed application discovery extension, we integrated it into our core project based on the high-quality of the extension in addition to the good usability and time saving aspect when instrumenting applications with *Kieker*. As the results of the the modularization process were not sufficient yet, we performed a second iteration featuring a first microservice architecture. More precisely, the iteration led to several independent deployable services bundled with inter-service communication handled via the message broker *Kafka* and requests from the frontend towards the backend are passed through our reverse-proxy in form of *NGINX*. Furthermore, we enhanced our development and build process towards a more collaborative manner. Unfortunately, we were not satisfied with the results of the second iteration, because some services were still very large and poorly maintainable. Thus, we needed to perform a further decoupling of them. Additionally, we recognized that our release management and CI processes, as well as our documentation, still needed to be improved. Consequently, with these drawbacks in mind, we performed a third iteration, after which we achieved a fully decoupled microservice architecture, consisting of a set of self-contained systems and well-defined interfaces in-between. The inter-service communication is still handled through *Kafka*. Additionally, we replaced our reverse-proxy with *Traefik* for handling requests from the frontend towards the backend. For the release management and documentation, we further optimized our CI pipeline regarding *Docker* images and supplemental (API) documentation for both developers and users.

B. Lessons Learned

The lessons learned while applying our *CORAL* approach to *ExplorViz* within three successfully performed iterations are summarized in the following. Performing software development for research is a challenging task, especially when a large number of inexperienced students is involved. In our experience, providing an extensible software architecture is a crucial task for open source (research) projects. Furthermore, extension points, i.e., interfaces, should be well-documented to ease the development of extensions. Additionally, the overall software development process should base on accessible documentation for all stages during the development for all collaborators. This includes documentation of the employed software architecture and the extension mechanisms, but should also cover best practices, hints, or lessons learned.

Regarding software quality, especially with respect to maintainability, we recommend the usage of software quality tools. Static analysis tools such as *Checkstyle* or *PMD* in combination with configured common code styles support developers directly while they code. This way, common programming flaws can be avoided in the committed source code and thus result in less bugs and required bug fixes. Also, we suggest the setup and usage of CI pipelines that allow a project to automate their testing, code quality checking, and software building. Thereby, the complete build cycle can be tested periodically. If the building is triggered by commits, developers also get an early feedback if something went wrong.

Providing *Docker* images of *ExplorViz* provides great value. Developers and users are able to use pre-configured images of our software for specific use cases, which may be based on different versions. This approach also eases the release management process and facilitates developers to test and adapt their extension to upcoming versions.

C. Future Work

In the future, we are planning to evaluate our finalized project, especially in terms of developer collaboration. Additionally, we plan to move from our CI pipeline towards a continuous delivery (CD) environment. Thus, we expect to further decrease the interval between two releases and allow users to try out new versions, even development snapshots, as soon as possible. Furthermore, we plan to use architecture recovery tools like [59] for refactoring or documentation purposes in upcoming versions of *ExplorViz*. Recently, we applied *ExplorViz* within case study, where we successfully performed a microservice decomposition with static and dynamic analysis of a monolithic application [60]. As a result, we plan to investigate, if we could enhance our *CORAL* approach with the applied decomposition process for future projects.

References

[1] C. Zirkelbach, A. Krause, and W. Hasselbring, "Modularization of Research Software for Collaborative Open Source Development," in *Proceedings of the The Ninth International Conference on Advanced Collaborative Networks, Systems and Applications (COLLA 2019)*, Jun. 2019, pp. 1–7.

[2] C. Goble, "Better Software, Better Research," *IEEE Internet Computing*, vol. 18, no. 5, pp. 4–8, Sep. 2014.

[3] A. Johanson and W. Hasselbring, "Software engineering for computational science: Past, present, future," *Computing in Science & Engineering*, vol. 20, no. 2, pp. 90–109, Mar. 2018. DOI: 10.1109/MCSE.2018.021651343.

[4] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," in *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017, pp. 243–246. DOI: 10.1109/ICSAW.2017.11.

[5] P. D. Francesco, P. Lago, and I. Malavolta, "Migrating Towards Microservice Architectures: An Industrial Survey," in *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, Apr. 2018, pp. 29–38.

[6] F. Fittkau, A. Krause, and W. Hasselbring, "Software landscape and application visualization for system comprehension with ExplorViz," *Information and Software Technology*, vol. 87, pp. 259–277, Jul. 2017. DOI: doi: 10.1016/j.infsof.2016.07.004.

[7] F. Fittkau, S. Roth, and W. Hasselbring, "ExplorViz: Visual runtime behavior analysis of enterprise application landscapes," in *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015 Completed Research Papers)*, AIS Electronic Library, May 2015, pp. 1–13. DOI: 10.18151/7217313.

[8] R. Heinrich, C. Zirkelbach, and R. Jung, "Architectural Runtime Modeling and Visualization for Quality-Aware DevOps in Cloud Applications," in *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017, pp. 199–201.

[9] R. Heinrich, R. Jung, C. Zirkelbach, W. Hasselbring, and R. Reussner, "An Architectural Model-Based Approach to Quality-aware DevOps in Cloud Applications," in *Software Architecture for Big Data and the Cloud*, I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, and B. Maxim, Eds., Cambridge: Elsevier, Jun. 2017, pp. 69–89.

[10] F. Fittkau, A. Krause, and W. Hasselbring, "Hierarchical software landscape visualization for system comprehension: A controlled experiment," in *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*, IEEE, Sep. 2015, pp. 36–45. DOI: 10.1109/VISSOFT.2015.7332413.

[11] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller, "Comparing Trace Visualizations for Program Comprehension through Controlled Experiments," in *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015)*, May 2015, pp. 266–276. DOI: 10.1109/ICPC.2015.37.

[12] Open Source Software Community, *Google Web Toolkit Project (GWT)*, version 2.8.2, last accessed: 2020.05.31. [Online]. Available: http://www.gwtproject.org.

[13] H. Al-Kilidar, K. Cox, and B. Kitchenham, "The use and usefulness of the iso/iec 9126 quality standard," in *Proceedings of the International Symposium on Empirical Software Engineering, 2005.*, 2005, pp. 126–132.

[14] Open Source Software Community, *Checkstyle*, version 8.10, last accessed: 2020.05.31. [Online]. Available: http://checkstyle.sourceforge.net.

[15] ——, *PMD*, version 6.10.0, last accessed: 2020.05.31. [Online]. Available: https://pmd.github.io.

[16] Eclipse Foundation, *AspectJ*, version 1.8.5, last accessed: 2020.05.31. [Online]. Available: https://www.eclipse.org/aspectj.

[17] Open Source Software Community, *H2*, version 1.4.177, last accessed: 2020.05.31. [Online]. Available: http://www.h2database.com.

[18] B. Upadhyaya, Y. Zou, H. Xiao, J. Ng, and A. Lau, "Migration of SOAP-based services to RESTful services," in *Proceedings of the 13th IEEE International Symposium on Web Systems Evolution (WSE)*, Sep. 2011, pp. 105–114.

[19] S. Vinoski, "RESTful Web Services Development Checklist," *IEEE Internet Computing*, vol. 12, no. 6, pp. 96–95, Nov. 2008, ISSN: 1089-7801.

[20] R. Kazman, M. Klein, and P. Clements, "Atam: Method for architecture evaluation," Carnegie-Mellon Software Engineering Institute, University Pittsburgh, PA, Tech. Rep., 2000.

[21] H. Koziolek, "Sustainability evaluation of software architectures: A systematic review," in *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, ser. QoSA-ISARCS '11, Boulder, Colorado, USA: ACM, 2011, pp. 3–12, ISBN: 978-1-4503-0724-6. DOI: 10.1145/2000259.2000263.

[22] J. Knodel and M. Naab, "Software architecture evaluation in practice: Retrospective on more than 50 architecture evaluations in industry," in *2014 IEEE/IFIP Conference on Software Architecture*, Apr. 2014, pp. 115–124. DOI: 10.1109/WICSA.2014.37.

[23] J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*. Springer, 2016.

[24] M. Dick and S. Naumann, "Enhancing software engineering processes towards sustainable software product design," in *Integration of Environmental Information in Europe*, K. Greve and A. B. Cremers, Eds., Aachen: Shaker Verlag, 2010.

[25] P. Clarke and R. V. O'Connor, "An approach to evaluating software process adaptation," in *Software Process Improvement and Capability Determination*, R. V. O'Connor, T. Rout, F. McCaffery, and A. Dorling, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 28–41, ISBN: 978-3-642-21233-8.

[26] C. Zirkelbach, A. Krause, and W. Hasselbring, "On the Modernization of ExplorViz towards a Microservice Architecture," in *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*, vol. Online Proceedings for Scientific Conferences and Workshops, Ulm, Germany: CEUR Workshop Proceedings, Feb. 2018.

[27] A. Tang, M. Razavian, B. Paech, and T. Hesse, "Human Aspects in Software Architecture Decision Making: A Literature Review," in *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, Apr. 2017, pp. 107–116.

[28] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An Empirical Study of Architectural Change in Open-Source Software Systems," in *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, May 2015, pp. 235–245.

[29] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An Empirical Study of Architectural Decay in Open-Source Software," in *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, Apr. 2018, pp. 176–17 609.

[30] H. Knoche and W. Hasselbring, "Drivers and barriers for microservice adoption – a survey among professionals in Germany," *Enterprise Modelling and Information Systems Architectures (EMISAJ) – International Journal of Conceptual Modeling*, vol. 14, no. 1, pp. 1–35, 2019. DOI: 10.18417/emisa.14.1.

[31] H. Knoche and W. Hasselbring, "Using Microservices for Legacy Software Modernization," *IEEE Software*, vol. 35, no. 3, pp. 44–49, May 2018, ISSN: 0740-7459.

[32] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*. Springer International Publishing, 2017, pp. 195–216.

[33] N. Alshuqayran, N. Ali, and R. Evans, "A Systematic Mapping Study in Microservice Architecture," in *Proceedings of the 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, Nov. 2016, pp. 44–51.

[34] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating Towards Microservices: Migration and Architecture Smells," in *Proceedings of the 2nd International Workshop on Refactoring*, ser. IWoR 2018, Montpellier, France: ACM, 2018, pp. 1–6.

[35] Oracle, *Jersey Project*, version 2.27, last accessed: 2020.05.31. [Online]. Available: https://jersey.github.io.

[36] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis," in *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, ACM, Apr. 2012, pp. 247–248.

[37] Ember Core Team, *Ember.js*, version 3.6.0, last accessed: 2020.05.31. [Online]. Available: https://www.emberjs.com.

[38] Joyent, *Node.js*, version 10.15.0, last accessed: 2020.05.31. [Online]. Available: https://nodejs.org.

[39] Open Source Software Community, *json:api*, version 1.0.0, last accessed: 2020.05.31. [Online]. Available: https://jsonapi.org.

[40] A. Krause, C. Zirkelbach, and W. Hasselbring, "Simplifying Software System Monitoring through Application Discovery with ExplorViz," in *Proceedings of the Symposium on Software Performance 2018: Joint Developer and Community Meeting of Descartes/Kieker/Palladio*, Nov. 2018.

[41] L. Garber, "Gestural Technology: Moving Interfaces in a New Direction," *Computer*, vol. 46, no. 10, pp. 22–25, 2013, ISSN: 0018-9162. DOI: 10.1109/MC.2013.352. [Online]. Available: http://dx.doi.org/10.1109/MC.2013.352.

[42] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring Software Cities in Virtual Reality," in *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*, 2015, pp. 130–134. DOI: 10.1109/VISSOFT.2015.7332423. [Online]. Available: http://dx.doi.org/10.1109/VISSOFT.2015.7332423.

[43] A. Elliott, B. Peiris, and C. Parnin, "Virtual Reality in Software Engineering: Affordances, Applications, and Challenges," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 547–550. DOI: 10.1109/ICSE.2015.191.

[44] C. Zirkelbach, A. Krause, and W. Hasselbring, "Hands-On: Experiencing Software Architecture in Virtual Reality," Kiel University, Research Report, Jan. 2019. [Online]. Available: http://oceanrep.geomar.de/45728/.

[45] NGINX, *NGINX*, version 1.15.8, last accessed: 2020.05.31. [Online]. Available: http://nginx.org.

[46] Apache Software Foundation, *Apache Kafka*, last accessed: 2020.05.31. [Online]. Available: https://kafka.apache.org.

[47] Open Source Software Community, *Spotbugs*, version 3.1.10, last accessed: 2020.05.31. [Online]. Available: https://spotbugs.github.io.

[48] ESLint Team, *ESLint*, version 5.12.0, last accessed: 2020.05.31. [Online]. Available: https://eslint.org.

[49] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 426–437.

[50] Open Source Software Community, *TravisCI*, last accessed: 2020.05.31. [Online]. Available: https://travis-ci.org.

[51] ——, *Sonatype*, last accessed: 2020.05.31. [Online]. Available: https://oss.sonatype.org.

[52] ExplorViz Team, *ExplorViz Developer and User Documentation Wiki*, last accessed: 2020.05.31. [Online]. Available: https://github.com/ExplorViz/Docs/wiki.

[53] Open Source Software Community, *Swagger*, last accessed: 2020.05.31. [Online]. Available: https://swagger.io.

[54] ——, *Traefik*, last accessed: 2020.05.31. [Online]. Available: https://containo.us/traefik/.

[55] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proceedings*

of the 10th Computing Colombian Conference (10CCC), Sep. 2015, pp. 583–590.

[56] 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 *Proceedings of the XLII Latin American Computing Conference (CLEI)*, Oct. 2016, pp. 1–11.

[57] J. Gouigoux and D. Tamzalit, "From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture," in *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017, pp. 62–65. DOI: 10.1109/ICSAW.2017.35.

[58] R. Chen, S. Li, and Z. Li, "From Monolith to Microservices: A Dataflow-Driven Approach," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 466–475.

[59] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems," in *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017, pp. 298–302.

[60] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice Decomposition via Static and Dynamic Analysis of the Monolith," in *Proceedings of the IEEE International Conference on Software Architecture (ICSA 2020)*, Mar. 2020. [Online]. Available: https://arxiv.org/abs/2003.02603.