Modularization of Research Software for Collaborative Open Source Development

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

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

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

***Abstract*—Software systems evolve over their lifetime. Changing conditions, such as requirements or customer requests make it inevitable for developers to perform adjustments to 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 describe the modularization measures and present how we solved occurring issues and enhanced our development process. Afterwards, we illustrate our modularization approach with our modernized, extensible software system architecture and highlight the improved collaborative development process. Finally, we present a proof-of-concept implementation featuring several developed extensions in terms of architecture and extensibility.**

***Keywords*–collaborative software engineering; open source software; software visualization; architectural modernization; 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. Open research software [1] 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 [2]. 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 [3]. 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 [4]. In this paper, we report on the modularization process of our open source research project *ExplorViz* towards a more collaboration-oriented development process on the basis of a microservice architecture. We later call the outdated version *ExplorViz Legacy*, and the new version just *ExplorViz*.

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 illustrate our software system and underlying architecture of *ExplorViz Legacy* in Section III. The following modularization and modernization process as well as the target architecture of *ExplorViz* are described in Section IV. Section V introduces our proof of concept in detail, including an evaluation based on several developed extensions. Our ongoing work in terms of achieving an entire microservice architecture is presented in Section VI. Section VII discusses related work on modularization and modernization towards microservice architectures. Finally, the conclusions are drawn and an outlook is given.

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 [5], [6]. 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 [7], [8] and experiments [9], [10]. The project is developed from the beginning on GitHub with a small set of core developers and many collaborators (more than 30 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 25 student theses as well as multiple projects contributed to *ExplorViz*. We initially chose the Java-based Google Web Toolkit (GWT) [11], 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.

1) Extensibility & Integrability: ExplorViz Legacy's concerns are divided in core logic (core), e.g., 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 feature branch.

2) 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. 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). 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. 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* [12] or *PMD* [13]. 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.

3) 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*.

III. *ExplorViz Legacy*

The overall architecture and the employed 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, which employs in the case of Java *AspectJ*, an aspect-oriented programming extension for Java [14]. *AspectJ* allows us to intercept an application by bytecode-weaving in order to 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 a *H2 database* [15] 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 based on HTTP. 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 hardwired 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 in order to move from a monolithic, to a distributed (web) application divided into separately maintainable and deployable backend and frontend components.

IV. Modularization Process and Architecture of *ExplorViz*

The previously mentioned drawbacks in *ExplorViz Legacy* and recent experience reports in literature about successful applications of alternative technologies, e.g., Representational State Transfer (REST or RESTful) Application Programming Interfaces (API) [16], [17], were triggers for a modularization and modernization. In [18], we gave a very brief description on the modernization process of *ExplorViz* towards a microservice architecture. 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 [18]. Furthermore, developers, especially inexperienced ones, tend to have potential biases during

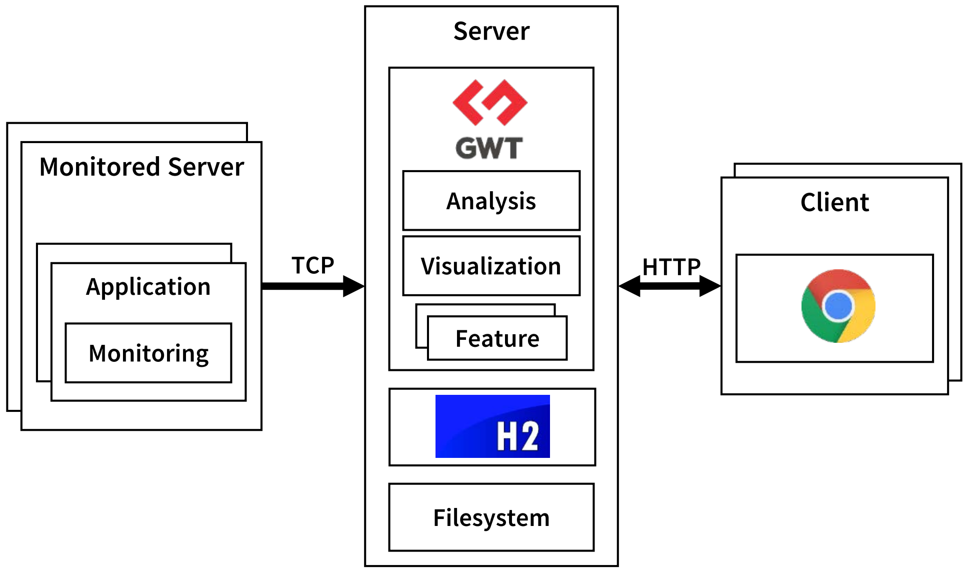

Figure 1: Architectural overview and software stack of *ExplorViz Legacy*.

the development of software, e.g., they make decisions on their existing knowledge instead of exploring unknown solutions [19]. A more detailed description of decision triggers and the decision making process will be published in a technical report [20]. In general, there exist many drivers and barriers for microservice adoption [21]. Typical barriers and challenges are the required additional governance of distributed, networked systems and the decentralized persistence of data.

As a result of this process, we agreed on building upon an architecture based on microservices as shown in Figure 2. This architectural style offers the ability do divide monolithic applications into small, lightweight, and independent services, which are also separately deployable [3], [22]–[24]. However, the obtained benefits of a microservice architecture can bring along some drawbacks, such as increased overall complexity and data consistency [25].

1) 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. This allows us to exchange the microservices, as long as we take our specified APIs into account. The backend is implemented as a Java-based web service based on the *Jersey Project* [26], which provides a RESTful API via HTTP for clients. Furthermore, we replaced our custom-made monitoring component by the monitoring framework *Kieker* [27]. 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. The frontend uses the JS framework *Ember.js*, which enables us to offer visualizations of software landscapes to clients with a web browser [28]. Since *Ember* 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* 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 [29]. 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* 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.

2) 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* [30] 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.

A. Software Configuration & Delivery

One of our goals was the ability to easily exchange 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.

V. Proof-of-Concept Implementation

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*. Both frameworks have a large and active community and offer sufficient documentation, which is important for new developers. As shown in Figure 2, we strive for an easily maintainable, extensible, and plug-in-oriented microservice architecture. Since the end of our modularization and modernization process in early 2018, we were able to successfully develop several extensions both for the backend and the frontend. Two of them are described in the following.

1) Application Discovery: Although we employ a monitoring framework, 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 [31], 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. Furthermore, this extension properly configures and manages the monitoring framework. 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.

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*.

2) 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 unknown software is still a potentially challenging and time-consuming task. For this extension, three students followed a new approach using Virtual Reality (VR) for exploring software landscapes collaboratively. They employed head mounted displays (HTC Vive and Oculus Rift) to allow the collaborative exploration of software in VR. They built upon our microservice architecture and employed WebSocket connections to exchange data to achieve modular extensibility and high performance for this real-time user environment. As a proof of concept, they conducted a first usability evaluation with 22 probands. The results of this evaluation revealed a good usability and thus constituted a valuable extension to *ExplorViz*.

VI. Restructured Architecture and new Process

Our modularization approach started by dividing the old monolith into separated frontend and backend projects [18]. 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 3 and the new collaborative development process is described below. As reported in Section V, the new architecture already improved the collaboration with new developers who realized new features as modular extensions.

1) Extensibility & Integrability: Frontend extensions are based on *Ember's* addon mechanism. The backend, however, used the package scanning feature of *Jersey* to include extensions. The result of this 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 [32], 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. The inter-service communication is now realized with the help of *Apache Kafka* [33]. *Kafka* is a distributed streaming platform with fault-tolerance for loosely coupled systems. The decomposition into several independent microservices and the new inter-service communication approach both facilitate low coupling in our system.

2) Code Quality & Comprehensibility: The improvements for code quality and accessibility, which were introduced in our first 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* [34], we impose their usage on contributors for Java code. For JS, we employ *ESLint* [35], i.e., a static analysis linter, with an *Ember* community-driven rule set. All tools are integrated into our continuous integration pipeline configured in *TravisCI* [36].

A. 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 integration of extensions, but their delivery was cumbersome. Due to the tight coupling at source code level we had to

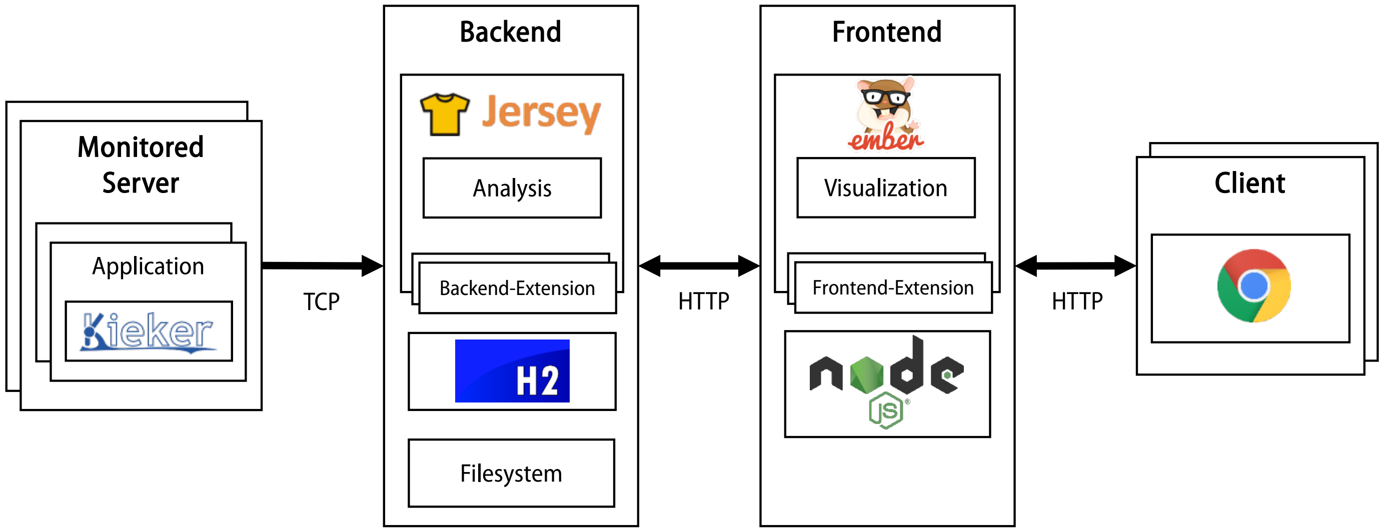

Figure 2: Architectural overview and software stack of the modularized *ExplorViz*.

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 3 we can now provide a jar file for each service with an embedded web server. This modern approach for Java web applications facilitates 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 the continuous integration 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* addon inside of a deployed *Ember* application. We are currently developing a build service for users that ships ready-to-use, pre-built configurations of our frontend. Users can download and deploy these packages. Alternatively, these configurations will also be usable as Docker containers.

VII. 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. [25] 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. [22] 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 [3], 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 re-build 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 are further planning to develop our pipeline towards continuous delivery for all microservices mentioned in Section VI to minimize the release cycles and offer development snapshots.

VIII. Conclusion

In this paper, we report 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 describe technical and development process related drawbacks of our initial project state until 2016 in *ExplorViz Legacy* and illustrate our modularization process and architecture. The process included not only a decomposition of our web-based application into several components, but also 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. We realized our modularization process and architecture in

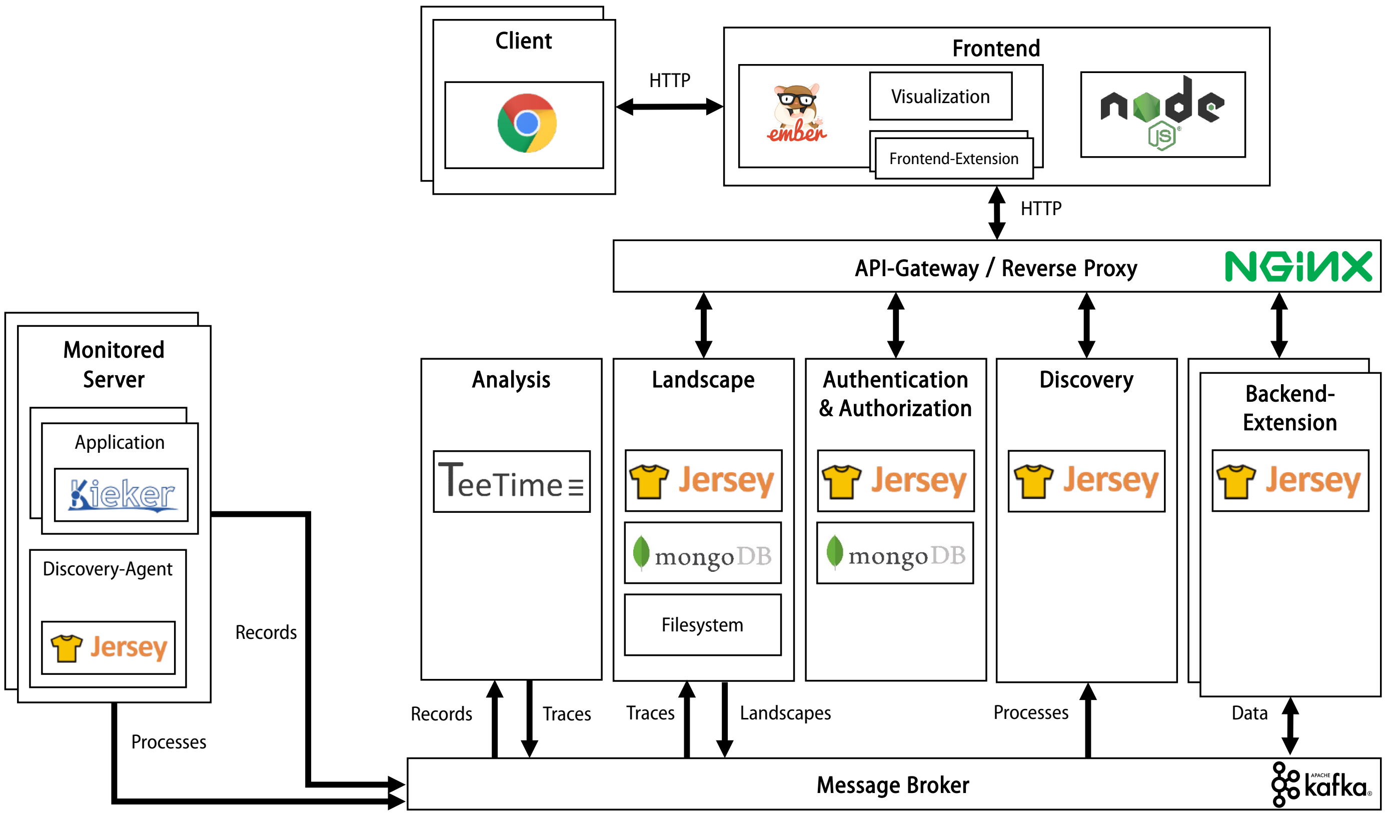

Figure 3: Architectural overview and software stack of the restructured *ExplorViz*.

terms of a proof-of-concept implementation and evaluated it afterwards by the development of several extensions of *ExplorViz*. However, the modularization process is not fully completed, as yet. We are still improving the project in order to achieve a fully decoupled microservice architecture, consisting of a set of self-contained systems and well-defined interfaces in-between. In the future, we are planning to evaluate our finalized project, especially in terms of developer collaboration. Additionally, we plan to move from our continuous-integration pipeline towards a continuous-delivery environment. Thus, we expect to 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 [37] for refactoring or documentation purposes in upcoming versions of *ExplorViz*.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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. 29–2909.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Open Source Software Community, *Google Web Toolkit Project (GWT)*, version 2.8.2, May 19, 2019. [Online]. Available: http://www.gwtproject.org.

[12] ——, *Checkstyle*, version 8.10, May 19, 2019. [Online]. Available: http://checkstyle.sourceforge.net.

[13] ——, *PMD*, version 6.10.0, May 19, 2019. [Online]. Available: https://pmd.github.io.

[14] Eclipse Foundation, *AspectJ*, version 1.8.5, May 19, 2019. [Online]. Available: https://www.eclipse.org/aspectj.

[15] Open Source Software Community, *H2*, version 1.4.177, May 19, 2019. [Online]. Available: http://www.h2database.com.

[16] 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.

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

[18] 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.

[19] 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.

[20] C. Zirkelbach, A. Krause, and W. Hasselbring, “On the Modularization of ExplorViz towards Collaborative Open Source Development,” Kiel University, Technical Report TR 1902, May 2019, to appear.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Oracle, *Jersey Project*, version 2.27, May 19, 2019. [Online]. Available: https://jersey.github.io.

[27] 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.

[28] Ember Core Team, *Ember.js*, version 3.6.0, May 19, 2019. [Online]. Available: https://www.emberjs.com.

[29] Joyent, *Node.js*, version 10.15.0, May 19, 2019. [Online]. Available: https://nodejs.org.

[30] Open Source Software Community, *json:api*, version 1.0.0, May 19, 2019. [Online]. Available: https://jsonapi.org.

[31] 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.

[32] Nginx, *Nginx*, version 1.15.8, May 19, 2019. [Online]. Available: http://nginx.org.

[33] Apache Software Foundation, *Apache Kafka*, May 19, 2019. [Online]. Available: https://kafka.apache.org.

[34] Open Source Software Community, *Spotbugs*, version 3.1.10, May 19, 2019. [Online]. Available: https://spotbugs.github.io.

[35] ESLint Team, *ESLint*, version 5.12.0, May 19, 2019. [Online]. Available: https://eslint.org.

[36] Open Source Software Community, *TravisCI*, May 19, 2019. [Online]. Available: https://travis-ci.org.

[37] 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.