Designing a Generic Research Data Infrastructure Architecture with Continuous Software Engineering

Nelson Tavares de Sousa, Wilhelm Hasselbring
Software Engineering Group
Kiel University
Kiel, Germany
{tavaresdesousa, hasselbring}@email.uni-kiel.de

Tobias Weber, Dieter Kranzlmüller
Leibniz Supercomputing Centre
Bavarian Academy of Sciences and Humanities
Garching, Germany
{weber, kranzlmueller}@lrz.de

***Abstract*—Long-living software systems undergo a continuous development including adaptions due to altering requirements or the addition of new features. This is an even greater challenge if neither all users nor requirements are known at an initial design phase. In such a context, complex restructuring activities are much more probable, if the challenges are not taken into account from the beginning. We introduce a combination of the concepts of domain-driven design and self-contained systems to meet these challenges within the system's architecture design.**

We show the merits of this approach by designing an architecture for a generic research data infrastructure, a use case where the mentioned challenges can be found. Embedding this approach within continuous software engineering, allows to implement and integrate changes continuously, without neglecting other crucial properties such as maintainability and scalability.

***Index Terms*—Microservice, Self-Contained System, System-oriented Architecture, Continuous Software Engineering, Research Data Management**

I. Introduction

Software systems with a heterogeneous set of stakeholders experience various challenges within their requirements engineering. The extent to which all stakeholders are unknown is a further factor which impedes an initial complete system specification. Long-living systems may also undergo a changing set of stakeholders and therefore shifting requirements. In continuous software engineering, this needs to be considered throughout all stages of a system's life span. For instance, beginning with the system specification, the design needs to be able to compensate for changing requirements at any given point in time. Continuous integration and deployment need to be implemented in a way that new or changed requirements can be integrated into the running infrastructure.

In the following, we will introduce an approach to meet these challenges. This approach is validated in a reference implementation for the project *Generic Research Data Infrastructure* (GeRDI). By abstracting and extrapolating the requirements from a limited set of stakeholders, we are able to extract different domains regarding the feature set of GeRDI whereby in turn each major feature is implemented as a distinct self-contained system (SCS)[1]. This allows us to be adaptable regarding the requirements set and also to benefit from different properties of both concepts, domain-driven design and self-contained systems. Additionally, through loose coupling we are able to integrate already existing software and services such as high-performance computing or cloud computing and storage. Furthermore, changes remain within affected self-contained systems and will not propagate to other self-contained systems.

In Section II we introduce the application domain for our reference implementation. Our approach to meet the mentioned challenges is presented in Section III with the resulting architecture. Section IV shows an implementation of one use case. Deployment and operation requirements derived by this approach will be discussed in Section V. Section VI presents our conclusions and provides an outlook to future work.

II. Application Domain

Data-intensive research requires appropriate management of the research data. However, present solutions for data storage often lead to inaccessible data silos, instead of providing research data in a findable, accessible, inter-operable and reusable (FAIR) way [1]. Various initiatives in this field target on reducing barriers for researchers to establish an efficient data management and processing for their research data. This focus on making data accessible and shareable reflects the key points of the *European Open Science Cloud* (EOSC) [2]. Apart from data, services which offer capabilities to process and analyze the data also need to be reusable and shareable to other researchers to increase not only the impact of their research efforts, but also to make the research process more efficient, transparent, and reproducible.

The project GeRDI aims to provide an infrastructure which fosters FAIR data practices and also supports researchers in their data-driven workflow [3]. This is realized by integrating different domain services seamlessly into one single infrastructure. The continuous involvement of nine research groups of different research domains into the development process allows us to determine specific workflows and their involved services to optimize the infrastructure for real-life use cases. As a reference, selected research cases of these research groups will be reimplemented and extended using the GeRDI infrastructure.

One research case is provided by the *Environmental, Resource and Ecological Economics* Group (EREE) of Kiel

[1]http://scs-architecture.org/

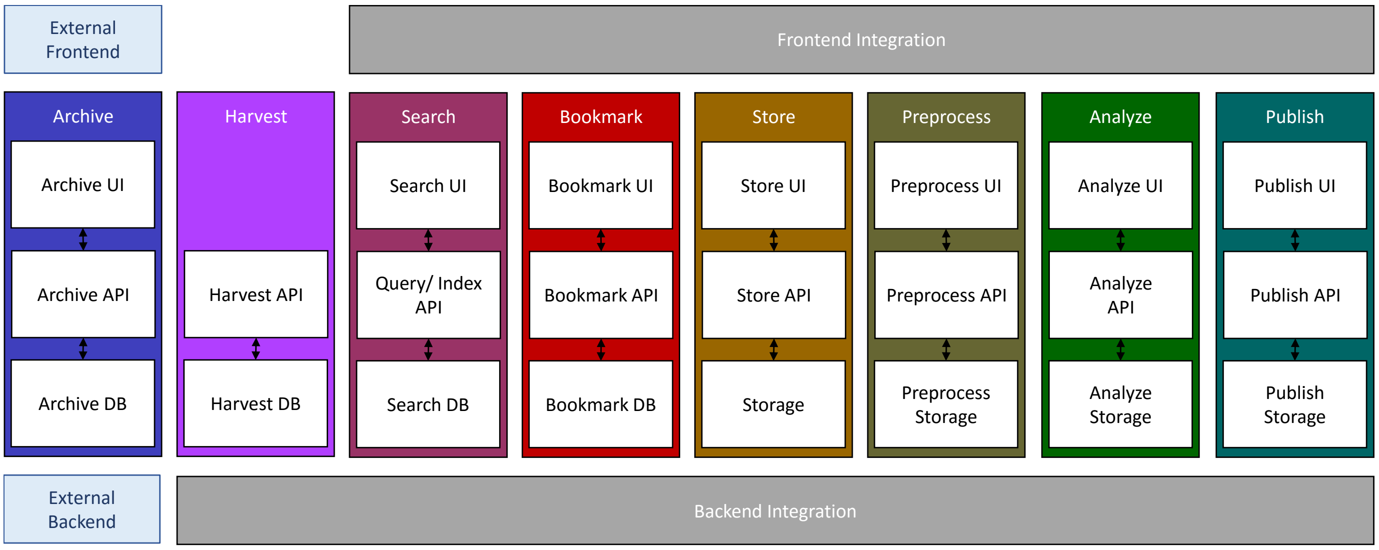

Fig. 1. The GeRDI Self-Contained Systems Architecture (a.k.a. Microservices Architecture)

University. In a report [4], published by the *WWF*, different economic and fishery management scenarios were evaluated in order to derive future changes in fishery catch rates. This research case illustrates a possible scientific workflow which GeRDI aims to support. Data is collected from multiple data repositories, aggregated and filtered in a preprocess step, and then passed to a computation model for scientific analysis and prediction of fishery catch rates. The other research groups contribute different research cases, to cover different research domains (such as digital humanities, hydrology, or socioeconomics) including different workflows and used services.

III. Architecture

Our goal is to design an architecture which allows us to react to changing requirements without major restructuring activities and which has a level of complexity that is as low as possible. To achieve this goal we rely on the strategy of domain-driven design (DDD) for the concept of our architecture. In DDD, complex systems are divided into bounded contexts in order to contain different domains within distinct components [5]. In our case, we derive different domains by clustering the required functionality of our research cases into sets of generic services. This is done by analyzing the workflows of all research cases and cutting them into different domains. As a result, we obtain a set of domains as shown in Figure 1. Each colored box depicts a service domain, enabling actions throughout the research data's life span. The service domains are all implemented as self-contained systems:

Archive depicts the data source which in most research cases is a research data repository for long-term data archival. An interface between such a research data repository and our infrastructure is realized through *Harvest* which collects the metadata, enriches it, and forwards it to a search index. With *Search*, a researcher can find relevant research data among all harvested, multidisciplinary research data repositories. A selection of relevant metadata for sharing search results or for further processing is then performed in *Bookmark*. After that, data is downloaded either to a local machine or a remote storage system (*Store*). The processing of data is divided into two stages. The first step is to normalize or pre-filter data in *Preprocess*. In *Analyze*, actual analysis on the preprocessed data is performed to gain new scientific insights. The new data can then be uploaded to a research data repository (*Publish*). This closes a cycle (not included in Figure 1) as the uploaded data is again available in the research data repository.

The required functionality can be provided as a specific implementation within each service domain. Additionally, the implementations of all domains are able to communicate with each other through remote interfaces. As a result, we are able to not only reimplement our individual services, but to also implement and integrate required functionality in the future, by implementing them with compliance to the given interfaces.

As mentioned in Section I, for the implementation of such an architecture, we make use of SCS as an architectural style. In our architecture concept, we vertically decompose the system along the domains and are therefore able to use methods of DDD not only as a design concept but also for the implementation of our architecture. A self-contained system depicts a certain functionality and implements it as a full stack with an user interface layer, business logic layer and persistence layer, which can be implemented as microservices [6]. Microservice architectures facilitate scalability [7], as well as agility and reliability [8]. Communication between SCS should be reduced to a minimum level. In cases where communication is inevitable, this should occur through well-defined REST-interfaces. Cross-cutting concerns regarding the implementation, such as an authentication and authorization infrastructure or system monitoring, are deployed within a backend integration layer.

A further layer for the frontend integration is also required,

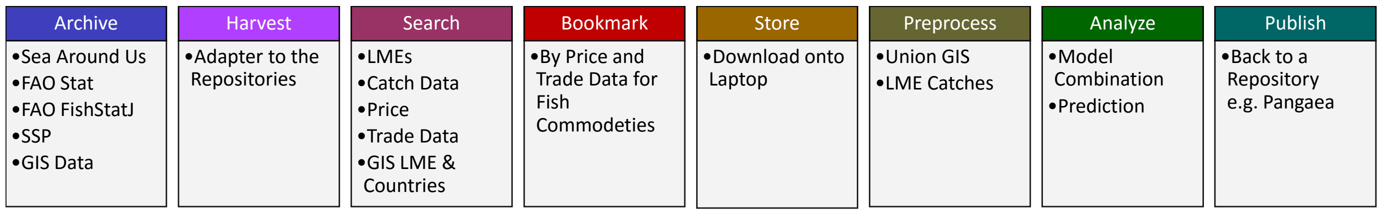

Fig. 2. Research Case Mapping to the GeRDI Architecture

as each SCS implements its own user interface. The SCS approach is scalable regarding different aspects [7]. The architecture scales well regarding its functionality, as new functions can be continuously implemented and integrated as a SCS. This is enabled by the inherent nature of loose coupling of SCS which makes them interchangeable. It scales well with the amount of developers, as each SCS can and should be developed by an individual team [8]. This allows to enable a community-driven infrastructure, as external developer teams may contribute their functions continuously to the GeRDI infrastructure. Due to the option of instantiating multiple instances of one SCS and a further load balancing in the frontend integration layer, this approach also introduces potential to scale well with regard to performance.

Figure 1 also illustrates how we implement the reference architecture of GeRDI. Each colored box depicts an SCS and therefore a domain of the complete system. White boxes within each SCS show the different layers of user interface (UI), business logic (API) and persistence (DB or Storage) within the SCS. Grey boxes at the top and bottom of the figure show both integration layers. As we do not re-implement research data repositories, their frontend and backend layers are not integrated into the GeRDI frontend. Additionally, implementations of *Harvest* do not require a UI, which leads to the lack of the corresponding microservice.

IV. Use Case Implementation

For the (re)implementation of research cases, this architecture allows to make use of existing middleware software wherever possible. Well-established software can be integrated into the infrastructure if it can be mapped to one domain and if it complies with its interfaces. Therefore, to implement complete research workflows, all required services need to be mapped first to the generic services model. Afterwards, each service is implemented as a SCS and integrated into the infrastructure.

Figure 2 shows a mapping of required services and/or functions for the EREE research case mentioned in Section II. We see in *Archive* the different research data repositories which deliver the relevant data. These repositories already exist and will be integrated into GeRDI by connecting them with different implementations of *Harvesters* for each repository. Both *Search* and *Bookmark* depict certain attributes for which data can be searched and bookmarked for. In our reference implementation for a search platform, we need to make sure we support the search and bookmarking for these attributes. Storage is provided by local computers in this use case. Therefore, the *Store* domain must support interfaces to download data to a local machine and to use the saved data for further steps. *Preprocess* and *Analyze* modify the original data. In a first step, data is aggregated by using geographic information system (GIS) data or catch rates of large marine ecosystem (LME). Then, by feeding the preprocessed data into a model, predictions for future scenarios regarding the catch rates can be made. Thus, both service domains rely on computation tasks. As a last step, the predicted data is again published to a research data repository, Pangaea[2] in this case.

As already mentioned, we will reuse existing software wherever possible in our reference implementation. The implementations for *Harvest* are newly developed. The *Search* makes use of Elasticsearch[3] as a search platform. For storage capabilities, network file sharing systems, such as Samba[4], can be used for this use case. By implementing a facade, this can be made GeRDI compliant. The computation steps require resources which can be provided by a cloud provider or a compute center. To enable the computation of the Matlab model, we reuse Jupyter[5] and integrate it in combination with its Matlab kernel into the infrastructure. For the publication of the newly generated data, we use Pubflow[6] which provides functionality to upload research data to repositories.

Different research cases may use different implementations. However, to benefit from a broader set of research data repositories, we encourage to make use of the same implementations for the service domains *Harvest*, *Search*, and *Bookmark*. The reference implementations for these domains will be open and accessible through a GeRDI portal.

V. Integration and Deployment

In this section we will briefly describe the requirements of an operational setup to run software as exemplified in Section IV.

The microservice architecture described in Section III necessitates a container-ready system to mirror the encapsulation. A registry is needed to disseminate the built images to the deployment contexts. Tagging the images is another requirement, since it facilitates the selection of compatible versions of the different SCS and enables the description of a

[2]https://www.pangaea.de/
[3]https://www.elastic.co/products/elasticsearch
[4]https://www.samba.org/
[5]https://jupyter.org/
[6]https://www.pubflow.uni-kiel.de/en

release manifest (i.e. a list of images/version pairs including the external dependencies).

After passing the code reviews and all tests within the continuous integration (CI) process the CI system builds the container image. This way the developers' assumptions with regard to the deployment context (e.g. available libraries) is encapsulated, thoroughly tested and ready to ship. The CI system needs to support this workflow.

We identified three deployment contexts: testing, staging and production. The testing context needs full automation, i.e. continuous deployment, and is used by developers to test and discuss features. Staging and production contexts are less automated since the robustness requirements are higher. They can therefore be classified as continuous delivery systems (i.e. manual work is necessary to deploy). Staging is not only used to prepare a release to the production context, but also as a preview for the stakeholders to facilitate an agile development approach. Several computing centers should be able to provide the computational resources to run all or parts of the three deployment contexts. At the same time some operational aspects need centralized services, such as monitoring and logging facilities. Some parts of the infrastructure (such as the search index) might profit from running on the same site to reduce performance penalties through network traffic. As a result, the deployment infrastructure needs to support fully-automated and semi-automated deployment workflows and allow for transparent integration of compute nodes, without losing the possibility to pin containers to specific nodes if necessary. In addition to that, scalability and availability requirements necessitate container orchestration abilities such as on-the-fly scaling, node draining, and rolling updates.

Since the deployment infrastructure is also developing over time, its setup needs to be documented and automated by a provisioning and configuration management system. The scripts and configuration for such a system are also part of the release process. Releases therefore consist of the source code, the container images, the setup scripts for the infrastructure and the release manifest. All release assets need to be available for the public (open source licenses).

The following setup meets the above described requirements and will be used for GeRDI:

- Containerization: Docker[7]
- Continuous Integration: Bamboo[8]
- Continuous Deployment/Delivery and container orchestration: Kubernetes[9]
- Provision and configuration management: Ansible[10]

In a recent literature review only 6 out 69 case studies were identified to discuss continuous practices in academic setups (cf. [9]). None of these describe the same requirements as pointed out in this section.

[7]https://www.docker.com

[8]https://de.atlassian.com/software/bamboo

[9]https://kubernetes.io

[10]https://www.ansible.com

VI. Conclusion & Outlook

We introduced an approach to handle an incomplete set of requirements through an appropriate architecture design. Our approach combines domain-driven design and self-contained systems to provide an infrastructure which can be used for the implementation of different and also unknown function requirements. With continuous software engineering we are able to continuously implement, deploy, and integrate functionality changes into a running system. The result is used for a generic research data infrastructure and allows to (re)implement existing and yet unknown use cases. As an example, we depicted one use case and introduced its implementation with this architecture. Challenges regarding the operation of such a system were also discussed and an appropriate setup was presented.

The development of GeRDI is in a early stage and therefore prototypical. Evaluations are required to show the benefits of this architecture in real-world usage. This includes the implementation of different use cases of other research domains which will show if the stated claims, regarding its adaptability, will hold.

Yet to be validated are other topics such as monitoring which is required for a useful system scaling and performance evaluation. The deployment and operation of an authentication and authorization infrastructure for such a system is an additional challenge of greater importance, due to a broader set of possible service providers.

Acknowledgements

This work was supported by the DFG (German Research Foundation) with the GeRDI project (Grants No. BO818/16-1 and HA2038/6-1).

References

[1] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific data*, vol. 3, 2016.

[2] C. H. L. E. G. on the European Open Science Cloud, "Realising the European Open Science Cloud," European Commision, Tech. Rep., 2013.

[3] R. Grunzke, T. Adolph, C. Biardzki, A. Bode, T. Borst, H.-J. Bungartz, A. Busch, A. Frank, C. Grimm, W. Hasselbring, A. Kazakova, A. Latif, F. Limani, M. Neumann, N. T. de Sousa, J. Tendel, I. Thomsen, K. Tochtermann, R. Müller-Pfefferkorn, and W. E. Nagel, "Challenges in Creating a Sustainable Generic Research Data Infrastructure," *Softwaretechnik-Trends*, vol. 37, no. 2, 2017.

[4] M. Quaas, J. Hoffmann, K. Kamin, L. Kleemann, and K. Schacht, "Fishing for Proteins," *WWF*, 2016.

[5] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[6] J. Lewis and M. Fowler, "Microservices," 2014, http://martinfowler.com/articles/microservices.html.

[7] W. Hasselbring, "Microservices for Scalability: Keynote Talk Abstract," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE 2016)*. New York, NY, USA: ACM, 2016, pp. 133–134.

[8] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Gothenburg, Sweden: IEEE, Apr. 2017, pp. 243–246.

[9] M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.