Migration from Monolith to Microservices with Legacy Compatibility

Daniel Haferkorn[a], Christian Kerth[b], Roland Rodenbeck[c], and Christian Zaschke[d]

[a,b,c,d]Fraunhofer IOSB, Fraunhoferstr. 1, 76131 Karlsruhe, Germany

ABSTRACT

Today's ISR (Intelligence, Surveillance and Reconnaissance) defense coalitions require storage and dissemination mechanisms that are able to cope with emerging changes to requirements and new features. Previous System of systems (SOS) architectures used to be built with years of planning, development, testing and deployment, usually in the form of distributed monoliths. Due to new requirements in ISR, shorter response cycles are required. To reach this goal, new approaches are of interest in the architectural style and workload sharing within the development team, resulting in the ability to better maintain and change existing software solutions. Ideally, such a shift results in improved scalability, replaceability, modularity and resilience.

In this context we examined our existing software that provides and also internally uses legacy middleware such as "Common Object Request Broker Architecture" (CORBA) (among others). The overall codebase was written in such a manner that it was easy to produce, i.e., technically motivated. The development team is rather small, so efficiency and the possibility to share (developer) knowledge is important.

Our goal was to evaluate the state of the art, thus being able to reasonably apply modern software development approaches that support mandatory legacy support. We attempted a restructuring of the codebase applying the principles of "Domain-Driven Design" with its "bounded contexts", resulting in domain-oriented source code that is easy to verify and maintain.

Keeping in mind our small development team, we aimed for shared responsibility, giving us the necessary resilience for unplanned staff absence.

In this publication, we present a possible migration path with its operational constraints (e.g., legacy interfaces) towards a more suitable software solution and the lessons learned during the process. In addition, we outline how this was achieved with a small headcount.

Keywords: migration, monolith, microservice, domain-driven design, system of systems architectures, legacy systems, ISR, agile

1. INTRODUCTION

Like systems in other domains, ISR (Intelligence, Surveillance and Reconnaissance) defense coalition systems require storage and dissemination mechanisms. Therefore the STANAG 4559, AEDP-17[1] defines a standard for interfaces for a storage and dissemination service. Our current implementation of this STANAG was designed as a System of systems (SOS) architecture. The result can be described as a set of distributed monoliths which communicate with each other over legacy interfaces. The implementation has been started several decades ago and while the underlying design was state of the art back then, the design can now be seen as outdated. This implementation is based on a technical design and not on a functional one.

Our colleagues addressed in their work "Evolution of the Coalition Shared Data concept in Joint ISR"[2] the origin of the data distribution that is subject of this publication. They also show the possible interoperability

Further author information: (Send correspondence to Daniel Haferkorn.)
Daniel Haferkorn: E-mail: Daniel.Haferkorn@iosb.fraunhofer.de https://orcid.org/0000-0002-2258-7784
Christian Kerth: E-mail: Christian.Kerth@iosb.fraunhofer.de https://orcid.org/0000-0002-9851-1751
Roland Rodenbeck: E-mail: Roland.Rodenbeck@iosb.fraunhofer.de https://orcid.org/0000-0001-9148-3658
Christian Zaschke: E-mail: Christian.Zaschke@iosb.fraunhofer.de https://orcid.org/0000-0002-6351-491X

within heterogeneous distributed systems in their publication "Interoperability of heterogeneous distributed systems"[3]. In the publication "Adaptation of interoperability standards for cross domain usage"[4] our colleagues discussed interoperability in the context of cross-domain data sharing.

Legacy requirements to the system are hindering modern development approaches. To achieve the possibility to cope with emerging changes to requirements and new features it is necessary to change our system fundamentally, while still maintaining interoperability with other STANAG 4559[1] implementations and the conformity to the standard. Due to new requirements in ISR, shorter response cycles are also required.

To reach this goal, new approaches are of interest in the architectural style and workload sharing within the development team, resulting in the ability to better maintain and change existing software solutions. Changing such an amount of legacy source code can be problematic even for larger development teams. Therefore, we need to find the right balance in the amount of concurrent changes to our architecture because our development team is rather small. Changes are not only necessary in the architecture but also in our way of working.

In order to make the system robust for future developments, it is planned to split the technical view into a domain-specific and, if possible, the monolith into microservices. First ideas for a microservice architecture are presented in the publication "Elaboration of a Domain Model for Migrating the Monolithic Software Architecture of a Data Management Server into a Microservice Architecture"[5]. The service should not simply be broken down into an arbitrary number of microservices, but a suitable cut should be found.

In the following section "Problem Description" 2 we discuss our situation. In the following section "Applying State of the Art Software Development" 3 we refer to how others have dealt with similar problems and consider their approaches to solving them. In the section "Goals" 4 we explain our goals which we want to reach by these changes. We show our initial steps and a possible execution in the section "Implementation" 5. In the last section "Conclusion" 6 we assess our initial work and give an outlook on possible future work.

2. PROBLEM DESCRIPTION

The composition of our developer teams is dynamic. Developers could be relocated from the current project team to another. On the other side, developers can also be allocated to the project at a later time or only just for a certain amount of time. This fluctuation leads to a situation where collective knowledge and available skills change over time. When a developer doesn't work on a specific project for a long time, his knowledge will also progressively get lost. In case a developer leaves the project team or even the whole department or company his implicit knowledge is completely lost if it hasn't been documented or otherwise transferred in an appropriate way. At the moment, it is quite challenging to compensate the absence of a team member and to distribute the workload over all developers.

The system considered in this paper is built up in a monolithic way. The system's interfaces are not under version control and have been defined in a very technical way instead of separating different functionalities from one another on domain level. As a consequence, the maintenance of the source code is rather expensive and it is difficult to log domain-orientedly to support problem analyses. Furthermore, it is not possible to replace single components or interfaces easily and without changing other parts of the software.

Currently, releases are only created on demand, hence rather infrequently (typically once or twice a year). This results in large version changes and long changelogs. The determining of possible breaking changes between two software versions is rather complicated due to the magnitude and complexity of changes involved.

The system's possibilities for vertical scaling[6] are limited. Even better performing hardware would only be able to cope up to a certain extent with larger amounts of requests that are made by clients on the software. The current architecture doesn't allow horizontal scaling, e.g., due to the use of stateful interfaces. However, we expect the need to support horizontal scalability in the near future.

In addition, the standard[1] on which the system is based on mandates an outdated middleware (CORBA) and supports web service-based interfaces only optionally. As not all implementations provide the optional webservice interfaces, their presence can not be relied upon in a typical deployment. Therefore, the CORBA middleware is required for legacy purposes. Overall, the standard is written and intended for a mainframe architecture instead of aiming at a modern architecture for lightweight services.

3. APPLYING STATE OF THE ART SOFTWARE DEVELOPMENT

In the literature and also in general the development process of software is getting faster and faster. This also allows short and precise feedback loops, such as those used in Scrum Sprints. Furthermore, more and more attempts are made to describe implementations in software development using the "* as code" concept. The reason for this is that it enables straightforward reproducibility and versioning. Such specifications like "configuration as code" also make it much easier to implement automation. Automation in turn leads to fewer human errors. Versioning, on the other hand, ensures that the current status of design decisions, requirements and documentation is available for each version. In addition, it is possible to store the artifacts and deployables for each build, which allows the developer to be much more involved in the overall development and deployment process. A continuous deployment also enables the possibility of monitoring changes in the software.

In the literature this process is usually described as continuous integration and either continuous delivery or continuous deployment (CI/CD), depending on the degree of automation. An overview can be seen in Figure 1. In addition, the term DevSecOps is increasingly used to put an emphasis on everlasting or ongoing security in relation to the cycle of development operation.

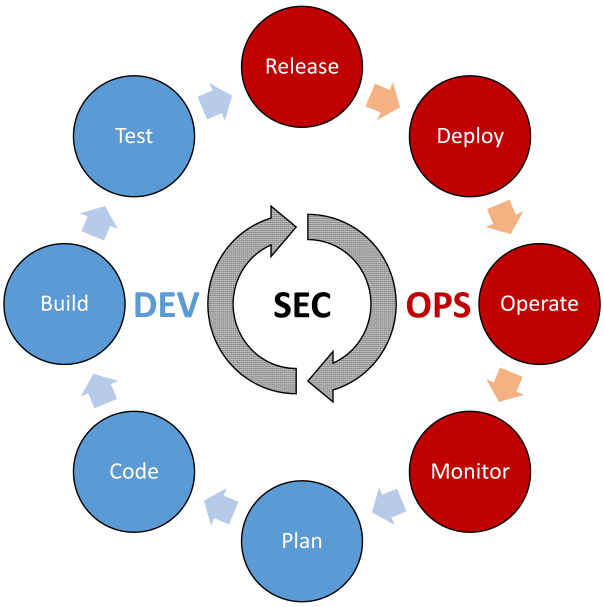

Figure 1. The DevSecOps cycle, see [7]

We practice the CI/CD process from the illustration as follows. The planning phase covers sprint planning and results in user stories based on requirements. Our source code and documentation is versioned and managed using Git and GitLab. The build process is automated using Jenkins and the project configuration is done using the build automation tool Maven. Domain-driven design (DDD)[8], behavior-driven development (BDD) and test-driven development (TDD) as well as Cucumber and Gherkin are used for the implementation. During the construction process, a static code analysis is performed using Sonarqube. The release at the end of the sprint as well as intermediate snapshot versions are stored in the Nexus artifact store as build artifacts. We also store our OSS dependencies in the Nexus, along with their license information, see [9] and [10].

In the future, the deployments are planned to be additionally deployed in service meshes and Kubernetes. This would make it possible to work and monitor directly on the latest version. A positive side-effect of the rapid deployment would also be that security problems would be found more quickly and could therefore be resolved faster.

When using microservices, it is important for developers to have a better integration of services and infrastructure services (logging, monitoring), as the number of services increases rapidly due to architectural changes.

An advantage of CI/CD for developers is the central availability of tools for every aspect of the process. That way, everyone can get insight into the full process at any time.

The current state of the art approach of CI/CD clearly shows that every team member should have a complete overview of the entire software development process. Faulty builds as well as the necessary steps to solve the underlying problems are visible to all team members. The fast feedback enables the possible use in agile work.

4. GOALS

Based on the documented problem and the described situation in relation to the state of the art, various goals arise. Currently, knowledge is bundled in singular competencies leading to multiple issues: if the employee with a specific skill set is not available, it is not possible to work on aspects involving this skill set. Thus, knowledge must be better distributed between developers. All developers should at least have enough basic knowledge in all parts of the software projects to distribute the work more evenly among all employees. This, together with adjustments to the software components and automation, should also reduce the maintenance effort throughout the projects and improve testability and also enable better proof of correctness. Another expected benefit is that both knowledge and employees should be more resilient towards outages. This means that knowledge is neither lost nor is there a standstill in development just because one or two developers with key competencies are not available. Overall, the agility of the team should improve and the team should be able to react better to changes. It is expected that the development cycle can also be accelerated.

Another goal is an improved (functional) modularity of the software components. A reasonable cut of the microservices according to suitable bounded contexts should be considered to avoid a too fine granularity of the microservices. This in turn results in a more modern software architecture while providing the interfaces required by the standard[1]. This should ultimately also lead to horizontal scalability[6]. The next section describes what we have done to achieve these goals as well as possible.

5. IMPLEMENTATION

Having identified the goals to work towards, we addressed them in prioritized order. We started by revising the waterfall[11] approach that dominated the work within the team up to now. This approach was replaced by a more agile one. To achieve the defined goals, basic elements of Scrum[12] such as sprints and retrospectives were added. Afterwards, additional elements of Scrum were analyzed and incorporated where they were deemed to support work efficiency.

When introducing new Scrum elements, it became obvious that communication within the team needed to be improved as well and that internal workflows had to be tailored towards Scrum and vice versa. E.g., part of the responsibilities of the Scrum Master were transferred to other roles due to the lack of available resources. On the one hand, communication between developers and project leads needed to be reviewed and refined. On the other hand, communication within the development team required enhancements. Through regular meetings, such as dailies, all participants are kept up-to-date with at least basic knowledge. Having daily meetings involving all team members bears the risk that too much detail is being communicated, resulting in both information overload and a lot of time being consumed unnecessarily. Also, there exists a certain risk that despite lengthy meetings not enough information could be exchanged. A happy medium needs to be identified where all members receive the amount of information they need, resulting in the possibility to identify issues timely. By giving immediate feedback and thus having short feedback cycles, a meaningful degree of abstraction can be achieved. Also, as a beneficial side effect, a common language emerges through incremental application of the approach.

In addition, workflows were established so that work packages can be shifted between team members more easily. A review mechanism in the software development workflow ensures that more than one person is aware of the modification made to the source code. This is ensured by using merge requests[13] for all source code changes, resulting in a four-eyes principle for every modification. In addition to much shorter time spans until issues can be identified, the merge requests result - over time - in a “group ownership of the source code”. A common understanding of “sufficiently clean code” can emerge among team members. By combining this with “definitions of done” for both individual work packages as well as the steps within the workflow, a common understanding of how work is conducted arises. Further, source code modifications are tested automatically

in an automated test environment that protects from unintentional regression of functionality or performance degradation of the software. This is complemented by checking the software against agreed software metrics and by setting quality gates[14] that prevent changes of poor quality to be merged into the main source code branch.

Another method to better share knowledge is the usage of event storming[15]. By applying this method, existing processes can be identified and visualized in their entirety. Also new processes can be fleshed out. Results of such event storming sessions can then be documented in an adequate form. Gained knowledge is being applied when developing the source code and when interacting with customers. Ultimately, this enables the developers to gain a T-shaped skill set[16], enabling versatile workload assignment.

The usage of event storming is accompanied by the introduction of domain-driven design (DDD)[8] and establishment of an ubiquitous language[17]. Such an ubiquitous language is then applied consistently in all software components, user frontends, documentation and the source code itself. This enables unprecedented consistency in the developed software and a reduced mental workload of involved people, as the number of synonyms and subtle differences in meaning decreases. With the help of DDD[8], bounded contexts[18] are being identified enabling a smaller and more concise decomposition of the resulting software. These bounded contexts were then implemented in prototypes, strengthening the gained knowledge in the development team. The internal software architecture of the implemented microservices was structured according to the principles of hexagonal architectures[8] in order to achieve a clean separation of the code domain logic and "technical adapter code". Having clean boundaries, both between the microservices and within them, enables writing better tests and isolating unwanted behavior more quickly.

The legacy interfaces can be expressed in the form of adapter microservices[19], i.e., they are being placed in front of the new, microservice-based interfaces to retain backward compatibility. By doing so, support for the legacy interfaces can be established while making it easier to prevent negative impacts on the newly established microservice landscape. Such adapter microservices would merely translate requests made through legacy interfaces towards microservices and make all information available in the microservice landscape through the interface of the standard[1].

For the dissemination of information between deployment nodes, new algorithms and data structures (Hash Tries)[20][21] were developed. These novel approaches enable strong guarantees for information sharing across deployment sites, but can also be used within a single deployment between the different microservices.

6. CONCLUSION

In this publication we first focused on the definition of the processes in teamwork and therefore addressed mainly the agile aspects. Therefore, we started to introduce Scrum in our team. We noticed that the standard Scrum-process can not be adopted one-to-one for us and has to be adapted in some places. This is partly due to the fact that we are a relatively small team and are not just doing software development. In addition, we have several software components in development which are to be maintained over several projects. The adjustments we had to make to Scrum relate to the scope of the Scrum Master role due to the lack of available resources. Therefore, some of the responsibilities of the role were transferred to other roles. We are in the process of evaluating these adjustments. It will become clear in the future how well this approach works. The advantages of better communication and basic common understanding already became directly noticeable. Moreover, the work is distributed better among all developers in the team. It has become evident to us that changing the tools and frameworks means that the work culture and way of working should also be changed.

There has also been initial work on the migration of our software components towards a microservice architecture. A first part which has been extracted is the interface and application logic for user authentication. This is an infrastructure service that can be used by several software components simultaneously. This implementation has already been successfully tested, has left the prototype phase and is now in productive use. However, other components still need to be converted. So we are still in the ongoing process of migration. Due to the already visible advantages and improvements, we are confident in going forward with the migration process.

In the future, the efforts to complete the migration to microservices will continue. Additionally, a technology refreshment in the standard[1] is planned using a published language[8] (OpenAPI[22], REST[23]). Furthermore, the operationalization of the prototypes, we have created so far over the course of the ongoing migration, shall be pushed forward. Containerization and the use in service meshes[24] should also be considered.

REFERENCES

[1] NATO Standardization Office (NSO), "NATO STANDARD ISR LIBRARY INTERFACE-AEDP-17." https://nso.nato.int/nso/nsdd/apdetails.html?APNo=2272 (2018). (Accessed: 15 March 2019).

[2] Essendorfer, B., Kerth, C., and Zaschke, C., "Evolution of the Coalition Shared Data Concept in Joint ISR," in [*Proceedings of the NATO IST/SET-126 Symposium on Information Fusion (Hard and Soft) for Intelligence, Surveillance & Reconnaissance (ISR), Joint Symposium IST-106 and SET-189*], (2015).

[3] Zaschke, C., Essendorfer, B., and Kerth, C., "Interoperability of Heterogeneous Distributed Systems," in [*Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security, Defense, and Law Enforcement Applications XV*], **9825**, 98250Q, International Society for Optics and Photonics (2016).

[4] Essendorfer, B., Kerth, C., and Zaschke, C., "Adaptation of Interoperability Standards for Cross Domain Usage," in [*Next-Generation Analyst V*], **10207**, 102070E, International Society for Optics and Photonics (2017).

[5] Zaschke, C., "Elaboration of a Domain Model for Migrating the Monolithic Software Architecture of a Data Management Server into a Microservice Architecture," in [*Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K Volume 3: KMIS, Vienna, Austria, September 17-19, 2019*], 212–218 (2019).

[6] Taherizadeh, S. and Stankovski, V., "Dynamic Multi-level Auto-scaling Rules for Containerized Applications," *The Computer Journal* **62**, 174–197 (05 2018).

[7] Robinson, M., "DevSecOps: A Complete Guide to What, Why, and How." https://www.plutora.com/blog/devsecops-guide (2020). (Accessed: 21 January 2020).

[8] Vernon, V., [*Implementing Domain-Driven Design*], Addison-Wesley, Amsterdam, 1. aufl. ed. (2013).

[9] Dyck, S., Haferkorn, D., and Sander, J., "An Organizational-Technical Concept to Deal with Open Source Software License Terms," *Journal of Systemics, Cybernetics and Informatics* **14**(3), 68–73 (2016).

[10] Dyck, S., Haferkorn, D., Kerth, C., and Schoebel, A., "Automating Open Source Software License Information Generation in Software Projects," *Journal of Systemics, Cybernetics and Informatics* **16**(5), 44–49 (2018).

[11] Royce, W. W., "Managing the Development of Large Software Systems," in [*Proceedings IEEE WESCON*], 1–9 (Aug. 1970).

[12] Schwaber, K. and Sutherland, J., "Der Scrum Guide," (2014).

[13] GitLab, "Merge Requests." Gitlab, https://docs.gitlab.com/ee/user/project/merge_requests/ (2020). (Accessed: 21 January 2020).

[14] SonarSource, "Quality Gates." SonarSource, https://docs.sonarqube.org/latest/user-guide/quality-gates/ (2020). (Accessed: 21 January 2020).

[15] Brandolini, A., "Event Storming." https://www.eventstorming.com/ (2020). (Accessed: 21 January 2020).

[16] Wikipedia, "T-shaped Skills." Wikipedia, https://en.wikipedia.org/wiki/T-shaped_skills (2020). (Accessed: 21 January 2020).

[17] Fowler, M., "Ubiquitous Language." https://martinfowler.com/bliki/UbiquitousLanguage.html (October 2006). (Accessed:21 January 2020).

[18] Evans, [*Domain-Driven Design: Tacking Complexity In the Heart of Software*], Addison-Wesley Longman Publishing Co., Inc., USA (2003).

[19] Brown, K. and Woolf, B., "Implementation Patterns for Microservices Architectures," in [*Proceedings of the 23rd Conference on Pattern Languages of Programs*], 1–35 (2016).

[20] Eastlake, D. and Hansen, T., "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," RFC 6234, RFC Editor (May 2011). http://www.rfc-editor.org/rfc/rfc6234.txt.

[21] Merkle, Ralph C., "A Digital Signature Based on a Conventional Encryption Function," in [*Advances in Cryptology — CRYPTO '87'*], Pomerance, Carl, ed., 369–378, Springer Berlin Heidelberg, Berlin, Heidelberg (1988).

[22] Initiative, O., "OpenAPI Specification." http://spec.openapis.org/oas/v3.0.2 (2020). (Accessed: 21 January 2020).

[23] Fielding, R. T., *REST: Architectural Styles and the Design of Network-based Software Architectures*, doctoral dissertation, University of California, Irvine (2000).

[24] Sharma, R. and Singh, A., [*Getting Started with Istio Service Mesh - Manage Microservices in Kubernetes*], Apress, New York, 1st ed. ed. (2019).