

Microservice Decomposition via Static and Dynamic Analysis of the Monolith

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

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

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

Stephan Lenga
Software Engineering Group
Kiel University
Kiel, Germany
stephan_lenga@web.de

Dan Krger
Solution Center Lottery Applications
adesso SE
Hamburg, Germany
dan.kroeger@adesso.de

Abstract—Migrating monolithic software systems into microservices requires the application of decomposition techniques to find and select appropriate service boundaries. These techniques are often based on domain knowledge, static code analysis, and non-functional requirements such as maintainability.

In this paper, we present our experience with an approach that extends static analysis with dynamic analysis of a legacy software system’s runtime behavior, including the live trace visualization to support the decomposition into microservices. Overall, our approach combines established analysis techniques for microservice decomposition, such as the bounded context pattern of domain-driven design, and enriches the collected information via dynamic software visualization to identify appropriate microservice boundaries.

In collaboration with the German IT service provider adesso SE, we applied our approach to their real-world, legacy lottery application *in|FOCUS* to identify good microservice decompositions for this layered monolithic Enterprise Java system.

Index Terms—microservices, architecture modernization, dynamic analysis, software visualization

I. INTRODUCTION

Improved maintainability, short time to market, and high scalability are some benefits of the microservice architectural style [1]. These advantages act as drivers for companies to modernize monolithic software systems towards microservices. While there are barriers for a microservice adoption [2], [3], they are an often desired architecture for migrating monolithic software systems to cloud-native environments [4].

These migration processes are rarely started as greenfield projects due to cost and time constraints [1]. Instead, monolithic software systems are incrementally decomposed into microservices. The decomposition, however, is a challenging task and requires many iterations to find suitable service boundaries [1], [5]. Industry best practices [6]–[8] and studies from academia [4] introduce strategies to support this task, but often share the same techniques, e.g., bounded contexts (BC) of the domain-driven design (DDD) [9], static code analysis, and refactoring based on non-functional requirements.

In this paper, we present our approach for microservice decomposition which additionally includes a dynamic analysis and visualization of a software system’s runtime behavior to find potential microservice boundaries. Furthermore, we report on its application to a real-world, legacy monolithic online lottery software called *in|FOCUS*.¹ *in|FOCUS* is a sales and management software solution for lottery providers that is developed by *adesso SE*, one of the largest IT service providers in Germany. It is provided as a software as a service (SaaS) solution which is currently used by several state lotteries.

Our approach for microservice decomposition starts with established migration processes including a domain analysis of *in|FOCUS* to familiarize with its ubiquitous language and business domain [5], [6]. Based on the outcome, we selected bounded contexts, i.e., the foundation for decomposing the layered monolithic Enterprise Java system into microservices when using DDD [1]. After gaining essential insights of *in|FOCUS*’ domain, we employed a static software structure analysis tool to map source code packages to the previously identified target boundaries. This enabled us to find overlaps among business functions for identifying ambiguities in the corresponding service boundaries [9], [10]. We then used dynamic analysis and live visualizations of *in|FOCUS*’ runtime behavior to refine previously identified service boundaries and find actual microservice candidates.

The remainder of this paper is structured as follows. Section II presents related work. Section III introduces the domain analysis of *in|FOCUS*. In Section IV, we build upon gathered domain knowledge and statically analyze *in|FOCUS*’ source code. Section V presents the procedure and results of the dynamic analysis of *in|FOCUS*’ runtime behavior. Finally, we discuss our main conclusions in Section VI.

¹<https://www.adesso.de/en/adesso-branch-solutions/lotteriegesellschaften/leistungen/loesungen/>

II. RELATED WORK

There exist several approaches, which are related to our migration process, based on the methodology or research topic.

Gysel et al. [11] utilize the static software system and domain analysis tool *Service Cutter*² to discover a suitable decomposition into microservices within the software architecture of the cargo tracking application *Cargo Tracker*.³ The decomposition process is based on several coupling criteria from literature and industrial experience. They extract coupling information from software artifacts of the software system like use cases and create an undirected, weighted graph to identify clusters. These clusters serve as candidates for the decomposition. Baresi et al. [12] also performed a tool-supported domain analysis on *Cargo Tracker*. In contrast to Gysel et al. they aim to provide an automated solution for decomposing a given domain into candidate microservices with the help of a pre-computed database of collections and similar words. The goal is to transform the domain into a machine-readable format to find cohesive groups. In comparison to both approaches, we focus on a collaborative approach with the software developers to create a context map, identify related BCs, and finally present potential candidate microservices. Additionally, these microservice candidates are verified and further decomposed due to applied static and dynamic analyses.

In [13], the authors present a process for decomposing an existing software asset into microservices. The process is based on experience from an industrial case study involving the migration of a COBOL legacy system towards Java. Their proposed modernization process consists of five steps, starting with defining external service facades and finally replacing service implementations with microservices. In contrast to our approach, we create a context map instead of a domain model and discuss use cases of the software system with the developers to identify BCs and microservice candidates.

A vertical decomposition of a monolith into several, self-contained systems is presented in [14]. The authors report on their successful decomposition and appropriate granularity of microservices as well as coupling, integration, scalability, and monitoring of microservices at the e-commerce platform *otto.de*. Compared to our approach, they focus on the technical and organizational aspects of the migration process and thus also address the development and deployment process. We employ a DDD-based approach for a microservice decomposition and do not (yet) focus on quality aspects of the migration.

In [15], the authors proposed a migration process for moving from a monolithic to a microservice architecture and applied it on a mobile learning application. Similar to our approach, they extracted microservice candidates from the original system based on DDD. In the next step, they determine, whether the database schema within the software system is consistent with the microservice candidates and the exclusion of inappropriate candidates. Afterwards, they extract source code related to the microservice candidates and establish a communication

between them. Finally the applied decomposition is tested and deployed in the execution environments. Compared to our approach, we additionally apply static and dynamic analysis to gather supplemental, valuable information of identified bounded contexts to optimize and further decompose our potential candidate microservices.

Our employed live trace visualization tool *ExplorViz* uses a 3D city metaphor visualisation for software applications. Related approaches employ this metaphor, for instance, for analyzing memory leaks [16] or synchronization problems [17]. Different to these approaches, we use the city metaphor for refining and revising the microservice decomposition obtained from static analysis.

III. DOMAIN-DRIVEN DESIGN

Fig. 1 depicts all actions, as well as intermediate and resulting artifacts of our analysis modeled as a Unified Modeling Language (UML) activity diagram. The domain analysis of *in|FOCUS* was divided in three phases, as indicated with the dashed boxes.

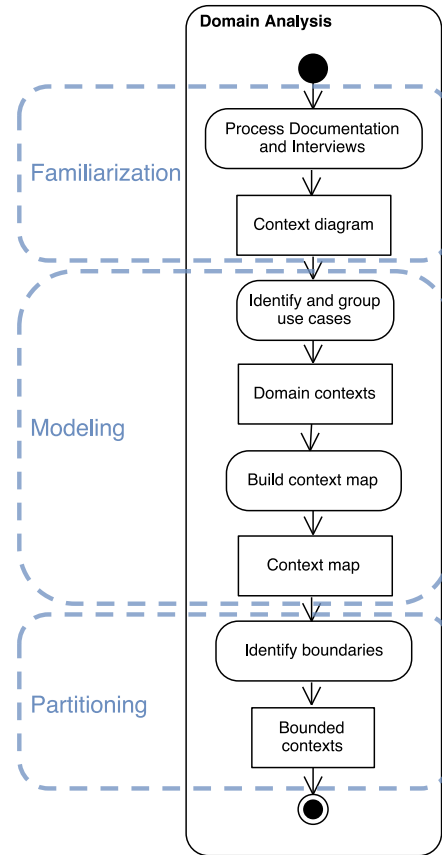


Fig. 1. Domain analysis procedure with actions (rounded rectangles), intermediate and resulting data (rectangles), and phases (blue boxes).

The *Familiarization* phase was used to gather information about *in|FOCUS* and to recover terms of its ubiquitous language. This was a crucial step, since we did not have any knowledge about the software, its structure, or behavior.

²<https://servicecutter.github.io>

³<https://cargo-tracker.gitbook.io>

For that reason, *adesso SE* supplied us with documentation, e.g., customer requirements specifications. Furthermore, we were able to interview developers and domain experts of *in|FOCUS*. A first result of this phase was a context diagram for the domain actors, i.e., *lottery application*, *user*, *customer*, and *state lottery*. *Customers* of *in|FOCUS* represent a subset of *users* who actively engage with the offered products. Remaining actors were summarized in a group called *other*, e.g., the OASIS system which logs every banned player to counteract gambling addiction. We used the actor context diagram in additional interviews to discover more of *in|FOCUS*' internals and behavior. The collected information of the *Familiarization* phase provided us with domain knowledge, which served as foundation for the next activities in the analysis.

The *Modeling* phase was used to clarify the behavior of *in|FOCUS* and derive an architectural overview of the system. With the help of developers and domain experts, we first identified the use cases of the *lottery application* actor towards the remaining actors. Then, we defined domain contexts, i.e., groups for related behavior or structure, and mapped the use cases onto these. We observed that some use cases related to multiple domain contexts. For example, the use case *transfer money to online wallet* was mapped both onto the *Payment Method* and *Online Wallet* contexts. This indicated a potential ambiguity, which was later reviewed in the static analysis (see Section IV).

The *Familiarization* phase revealed relationships between actors. Since the actors were related to the use cases and these were mapped to domain contexts, we proceeded by denoting the relationships at domain context level. The resulting context map with the defined domain context and their relationships can be seen in Fig. 2.

In the final *Partitioning* phase, we used the collected domain knowledge to identify target boundaries in the context map. These boundaries represent scopes in which each domain term has a unique definition, i.e., bounded contexts. Bounded contexts are the foundation for the microservice decomposition process [1], [18]. Equipped with the multiple domain context conflicts of the use cases in the *Modeling* phase, we were able to validate our chosen boundaries and resolve potential ambiguity at implementation level via static analysis (see Section IV).

IV. STATIC ANALYSIS

After decomposing *in|FOCUS* at the conceptual domain level, the next step was to partition its source code. For that, we employed the static software structure analysis tool *Structure101*⁴ to analyze the source code packages (SCP). *Structure101*'s leveled structure maps enabled us to observe dependencies among the SCPs and identify their locations at implementation level. We manually mapped the SCPs to the identified use cases of the *Modeling* phase (see Sec. III). Since we already assigned the use cases to the domain contexts (see Sec. III) as well as partitioned them into bounded contexts

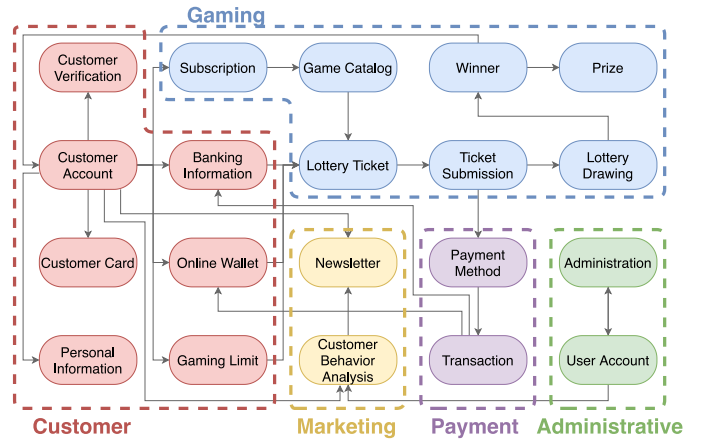


Fig. 2. Partitioning of the context map based on suitable boundaries (in color and dashed boxes) results in bounded contexts. A single rounded rectangle represents grouped use cases, i.e., a domain context.

(Fig. 2), we obtained a direct mapping of the *in|FOCUS* SCPs to the selected bounded contexts. This led to a first decomposition of the *in|FOCUS* application. Fig. 3 illustrates the assignment of the SCPs to the bounded contexts.

The *Customer* context contains the main functionality of the customer account management that is provided by the package *usermanagement*. This includes the account creation, login and logout, as well as editing personal information. For these use cases, identity verification services and external OASIS checkups are made accessible by the SCPs *services* and *externalservices*. A customer card contains the required customer account information for offline gaming at local lottery offices. The games played with the customer card are then booked on the online customer account. However, a customer card can exist without ever being used for gaming and can be seen as part of the customer account. Thus, we assigned the customer card management to the *Customer* context, rather than to the *Gaming* context. Since ordering and paying for a new customer card is also part of the customer card management, certain functions of the *subledger* package are also part of the *Customer* context. To complete the payment process, the *Customer* context is dependent on the *Payment* context.

The *Gaming* context covers the gaming functionality of *in|FOCUS*. The customer chooses a lottery game from an offered game catalog, fills out a lottery ticket and creates a game order (*gameprocessing*). The assignment to the customer account creates the indicated dependency between *Gaming* and *Customer*. *in|FOCUS* also provides instant lotteries such as online scratch tickets (*instantlottery*). These instant lotteries are immediately drawn. The conventional lottery games however are drawn at a certain point in time. Thereafter, the drawing results are imported (*prizedataimport*) and the individual lottery prize of each winner is calculated (*prizeanalyzer*). Alternatively, the customer can also subscribe to specific games (*tsubscriptions*). A subscribed game is repeatedly played for a defined period of time.

⁴<https://structure101.com/>

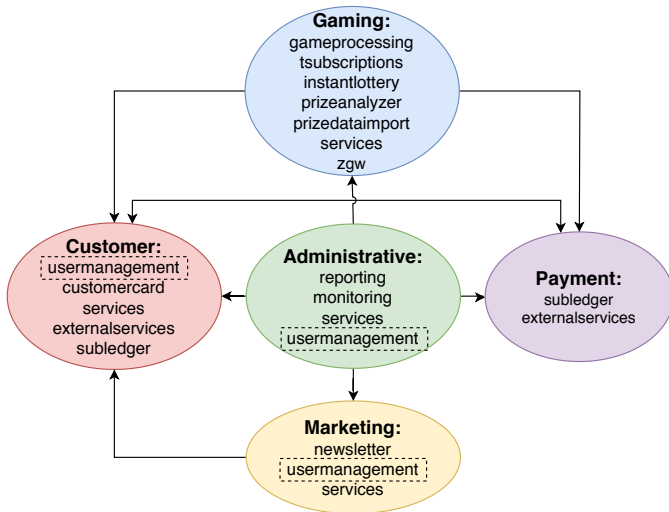


Fig. 3. As a result of the static analysis, source code packages were mapped to bounded contexts (bold). Dashed boxes indicate remaining ambiguities at implementation level.

Of course, the lottery tickets and the subscriptions have to be paid for. Therefore, the gaming process is dependent on the *Payment* context. The same dependency can be found in the package *zgw*, which is a German acronym for central winnings management. It manages the customer winnings. Winnings are either transferred back to the personal online wallet or to the bank account of the customer.

The *Payment* context handles and takes account of all incoming and outgoing money transactions (*subledger*). Additionally, it provides various payment methods to the customer (*externalservices*). To receive the up-to-date banking information or the online wallet ID of the customer, *Payment* is dependent on *Customer*.

The *Marketing* context provides newsletters to which the customers can subscribe. As long as the customer is subscribed to a newsletter, a notice, either through the website or via e-mail (*services*), is delivered. Lottery operators can create or edit newsletters. To promote these newsletters to potentially interested (new) customers, they can define a target group to which the newsletter is presented more vividly. This promotion process creates the dependency between *Marketing* and *Customer*.

The *Administrative* context comprises the reporting and monitoring services of *in|FOCUS*. The customer activity as well as the performance and the load of the software system are monitored. System reports are created continuously and distributed to the appropriate authority (*services*). Furthermore, this context manages the user accounts and their rights (*usermanagement*). The monitoring of the systems activity creates the dependencies to all other contexts of *in|FOCUS*.

After mapping the SCPs to the bounded contexts, we were able to validate our chosen boundaries and to identify potential ambiguities at implementation level. The domain context conflicts of the use cases in the *Modeling* phase (see Sec. III) already indicated required restructurings of some

boundaries. One example for this can be seen in Fig. 3: *usermanagement* was mapped to multiple bounded contexts due to shared code. This disclosed a required restructuring of this package to achieve well-defined service boundaries for a good microservice decomposition [1].

Fig. 3 revealed another problem: both, the *Customer* and the *Gaming* contexts were dependent on the *Payment* context. This could become a problem, if we intended to derive microservices from the bounded context-based decomposition. For example, a customer could not successfully acquire a customer card or play a lottery game when the microservice for the *Payment* context (also called *Payment service*) is unavailable. One could argue that buying a customer card is not one of the primary functionalities and that a temporary system failure is tolerable. Therefore, it would be acceptable to asynchronously forward the buying request to the *Payment* service. However, this approach would be inappropriate when it comes to playing lottery games. In this case, it would be preferable to successfully play a game without any dependencies to other services.

Fig. 4 presents four identified architectural alternatives to address this problem. Figure 4(a) depicts the asynchronous communication solution. The rationale for this approach is the redefinition of success for the use case of filling out and submitting a lottery ticket in the *Gaming* context. Beforehand, it was considered to be completed after the game order of the lottery ticket was created and paid for. As an alternative, we can consider the use case for the *Gaming* context to be completed right after the game order has been created. Completing the payment process is then part of the *Payment* context. The game order fails when the payment process has not been completed until the ticket submission deadline of the lottery game. In this case, the *Payment* service informs the *Gaming* service. Therefore, an asynchronous communication protocol between *Payment* and the other contexts or services would suffice. As a result, the dependencies between the services do not change. However, each service can now independently handle its own use cases.

Due to space limits for this paper, we only explain the further investigation of the finally chosen solution (a) in detail in the following Section V. The other solutions were eventually declined due to service granularity (b and c) and synchronization (d) problems.

V. DYNAMIC ANALYSIS

We expanded on the results of the static analysis with the help of our trace visualization tool *ExplorViz*.⁵ *ExplorViz* enables a live monitoring and visualization of large software landscapes [19]. In particular, the tool offers two types of visualizations – a landscape-level [20] and a 3D 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

⁵<https://www.explorviz.net/>

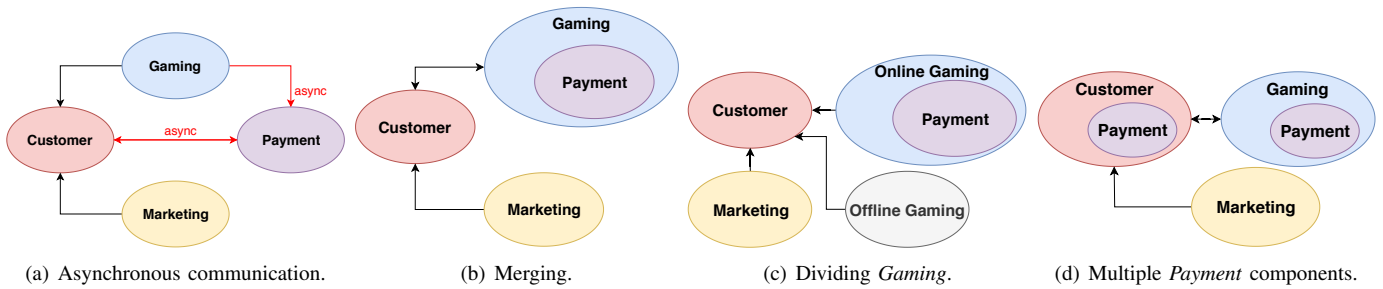


Fig. 4. Four architectural alternatives for addressing the dependency problem with the *Payment* context from Fig. 3.

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. It has been empirically evaluated via controlled experiments [21], [22]. *ExplorViz* started in 2012 as a layered, monolithic web application. Meanwhile, we modularized *ExplorViz* itself into a microservice architecture for improved collaborative open-source development [23]. In the present paper, we employ the 3D application-level perspective of *ExplorViz* for modularizing *in|FOCUS* into a microservice architecture.

By dynamically analyzing the behavior of *in|FOCUS* with *ExplorViz*, we refined the bounded contexts. This analysis of *in|FOCUS* has the following two goals: First, we further investigated the presented asynchronous solution for the *Payment* problem (see Figure 4). Second, we intended to discover additional microservice candidates within the software architecture to further decompose the resulting software architecture (see below).

Figure 5 presents an overview snapshot of *in|FOCUS*' runtime behavior visualized with *ExplorViz*. *ExplorViz*' 3D visualization is based on the city metaphor. SCPs are depicted as green boxes (❶). Classes are visualized as purple bars (❷), whereby the height of a class is related to the instance count at runtime. Communication between these entities is shown by orange lines (❸). The width of a communication line is related to the number of requests in that visualized snapshot. *ExplorViz* provides a timeline (❹), such that users can go back to previous visualized runtime behavior snapshots.

We continued the investigation of the asynchronous solution to the *Payment* problem as shown in Figure 4(a). The goal was to achieve decoupling via asynchronous communication between the *Customer* and *Gaming* services and the *Payment* service. Since the current definition of the service boundaries via bounded contexts lead to tight coupling of the three mentioned services when it comes to buying a product of the lottery application, we needed to redraw these boundaries. Figure 6 shows the runtime behavior when a customer transfers money from his registered bank account to his online wallet. This use case is related to all three services. The execution can be broken down into seven distinct steps:

Step 1: These classes (❶ in Figure 6) are the centerpiece of this use case and manage the payment process.

Step 2: The required permissions for transferring money to the target online wallet are checked (❷ in Figure 6).

Step 3: Checks of user rights and status are conducted in the *user* package (❸ in Figure 6).

Step 4: The *customer* package (❹ in Figure 6) delivers the required information on the customer and the associated online wallet.

Step 5: Inside the *workflow* package (❺ in Figure 6), the processing of the selected payment method is managed.

Step 6: The *businessprocess* package (❻ in Figure 6) is responsible for managing the appropriate business process. For that, business records are created to track the respective process.

Step 7: The account of the customer online wallet, onto which the money is transferred to, is managed (❼ in Figure 6).

To reduce the dependencies between *Customer* and *Payment* while enabling a feasible asynchronous communication between these services, we redefine the *Payment* service into an *Order* service. This service handles the payment process as well as the shopping cart which contains the items to buy. The shopping cart was previously part of the *Customer* service. This previous assignment of the shopping cart, however, led to unwanted communication between the *Gaming* service when buying lottery tickets. With the new assignment of the shopping cart component, both the *Customer* and *Gaming* services signal the *Order* service asynchronously to put a requested product into the cart. This can be done through choreographed publish and subscribe events. This approach has the advantage that all products can still be viewed or requested to buy even when the *Order* service is not available at that time. Furthermore, no more inter-service communication has to take place to complete the payment process. The other services can then be informed via asynchronous communication to confirm the successful execution of the payment process.

An important property of microservice architectures is that each microservice manages its own data store, possibly with different database technologies (polyglott persistence [24]). Consequently, the next step of our analysis process was to investigate a possible partition of *in|FOCUS*' data model.

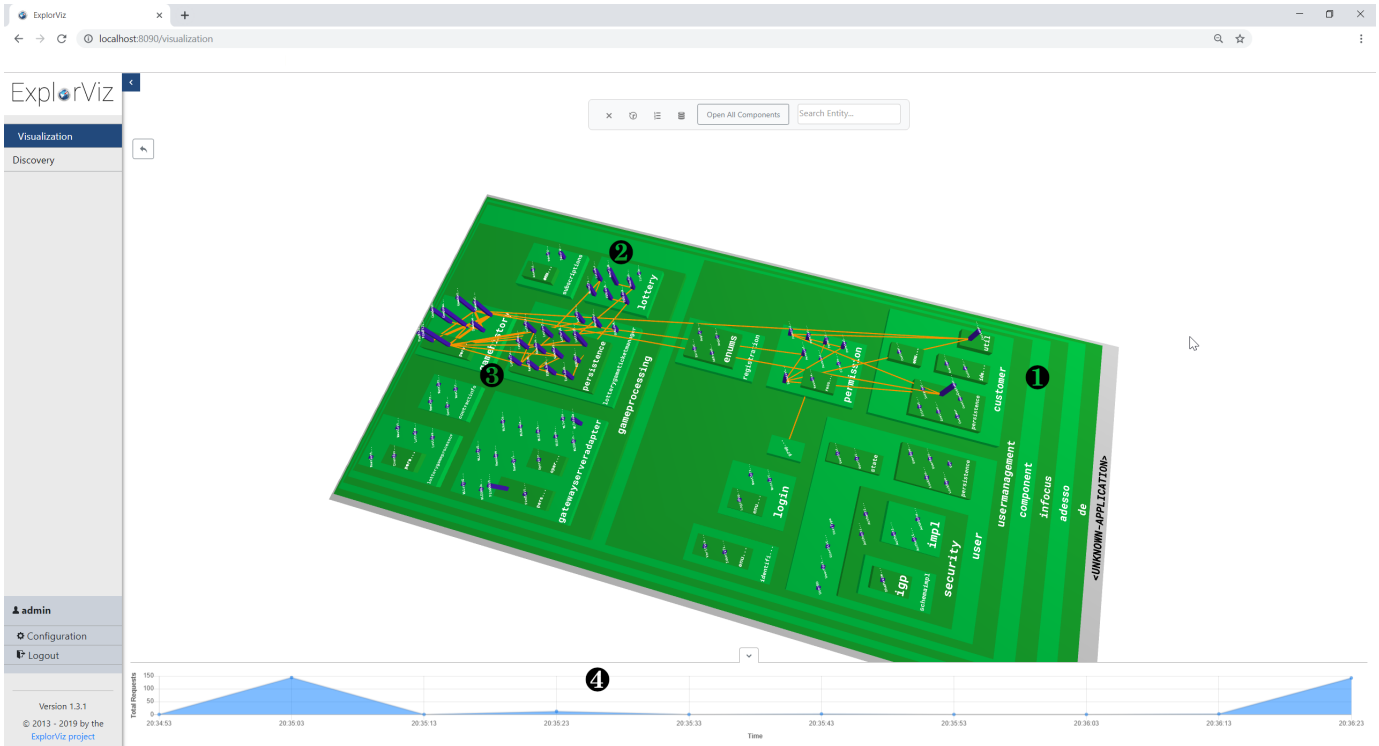


Fig. 5. Overview snapshot of an excerpt of *in|FOCUS*' runtime behavior with the *ExplorViz* 3D visualization.

Therefore, the centralized data governance had to be replaced by a distributed data model. For *in|FOCUS*, we statically analyzed the data model alongside previously identified use cases. We employed the database administration tool *DBeaver*⁶ to identify the database tables which are used by the use cases. Then, we mapped the tables onto the respective bounded contexts. This investigation of transaction boundaries and the resulting assignment provided us with an individual schema for each service [1]. Afterwards, the tables were adjusted to their context. Some tables, however, were used by multiple services. One example for that can be seen in Figure 7. The previous *User* table of the legacy system was used by multiple business functions, even though the definition of a user depends on its business context. For example, in contrast to the *Customer* service that considers users as customers, the *Gaming* service considers users as players of a game. These different views are now reflected in the separate data models for each bounded context in Figure 7.

To discover additional microservices, we executed the identified use cases and analyzed the resulting 3D visualization. Figure 8 shows an area of the runtime behavior of *in|FOCUS* when a lottery ticket is selected by a user and completed by the application. This use case can be divided into four distinct steps:

Step 1: Inside the *gatewayserveradapter* package (1 in Figure 8), a lottery manager is invoked for the appropri-

ate lottery game which was selected by the customer. Required lottery information, such as drawing dates, are collected and the lottery ticket is created. Furthermore, the delivery of the required lottery ticket information is done by the internal functionality of the persistence sub-package to assemble the actual ticket. These attributes can represent different available jackpots or gaming options for the selected game.

Step 2: The classes of the *lotterygameprocessor* package (2 in Figure 8) manage the gaming process of the selected lottery game.

Step 3: (not shown) Customer data, such as the ID and the personal gaming limits, are obtained. These are required for successfully playing a lottery game as customer.

Step 4: (not shown) The *lottery* package manages the lottery ticket. It tracks the selected fields and specific gaming strategies of the played lottery ticket.

We can see that the use case can be split into two main phases. First, the selected lottery ticket is assembled by collecting the gaming information. Thereafter, the customer fills out the ticket and selects different options for his game. Here, an opportunity can be found to define a microservice. This microservice, which we call *TicketManager*, handles the management and the composition of lottery game attributes for the creation of a specific lottery ticket. Hence, it holds all the available options for each lottery game. Depending on the customer's choice, it assembles the necessary information for creating a lottery ticket that the customer can then fill out. Therefore, the *TicketManager* microservice encapsulates the

⁶<https://dbeaver.io/>

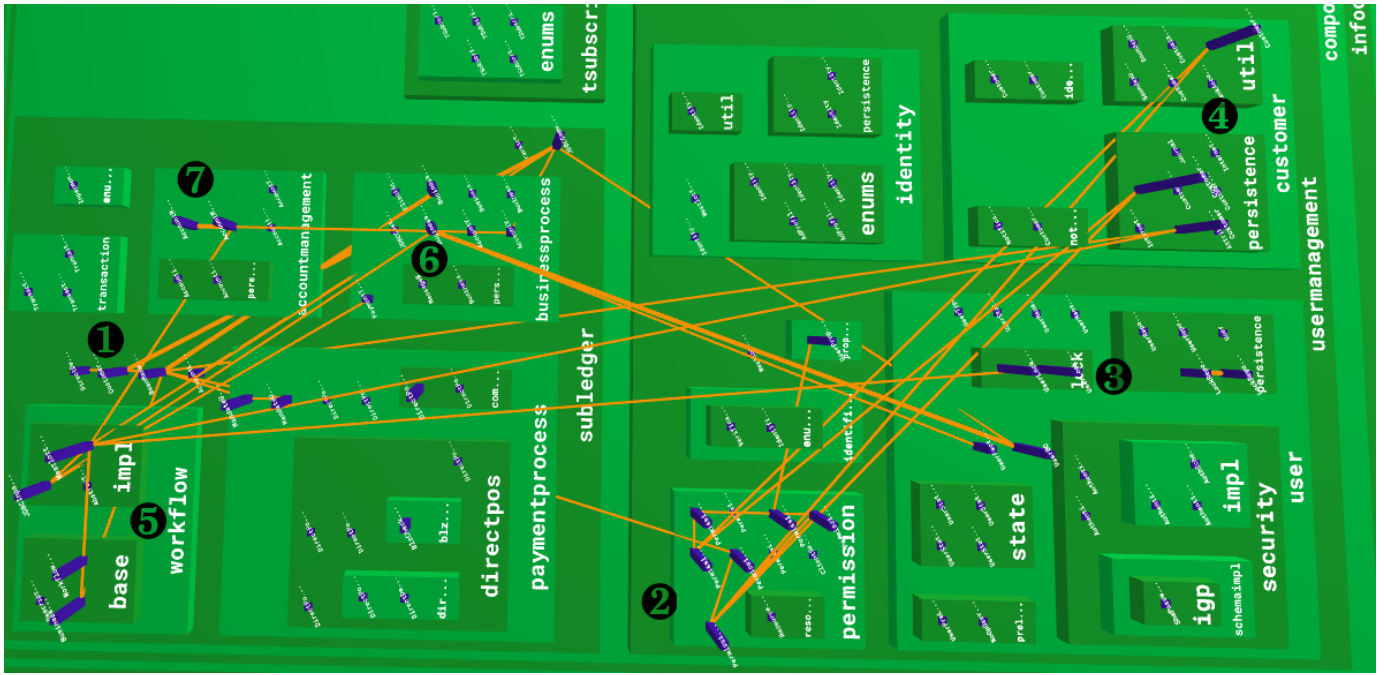


Fig. 6. Monitoring and visualizing *in|FOCUS'* runtime behavior with *ExplorViz* when a customer transfers money to his online wallet.

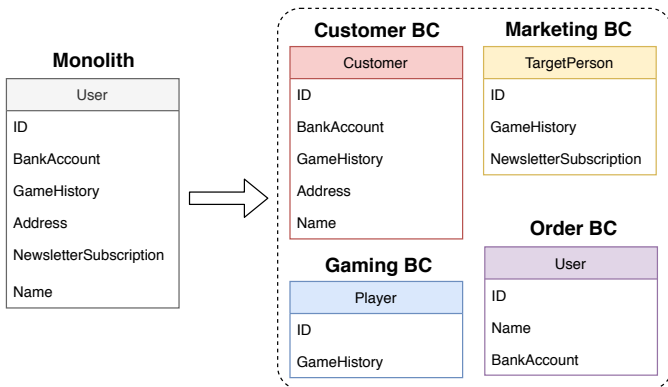


Fig. 7. Partition of the monolithic data model *User* into separate data models for each bounded context. The *Payment* bounded context is replaced by the *Order* bounded context.

functionality of the first step (❶ in Figure 8). The introduction of this microservice enables the developers to easily define and add new games to the application. Furthermore, this part of the gaming process can be seen as a potential bottleneck, depending on the simultaneously requested games. Therefore, the now achieved option of horizontal scaling is desirable. The *TicketManager* service additionally needs to submit the assembled attributes to the *Gaming* service which handles the further gaming process. One possibility is the asynchronous communication between these services by publishing and subscribing to events.

VI. CONCLUSIONS

In this paper, we present experience with our migration approach that extends static analysis with dynamic analysis of

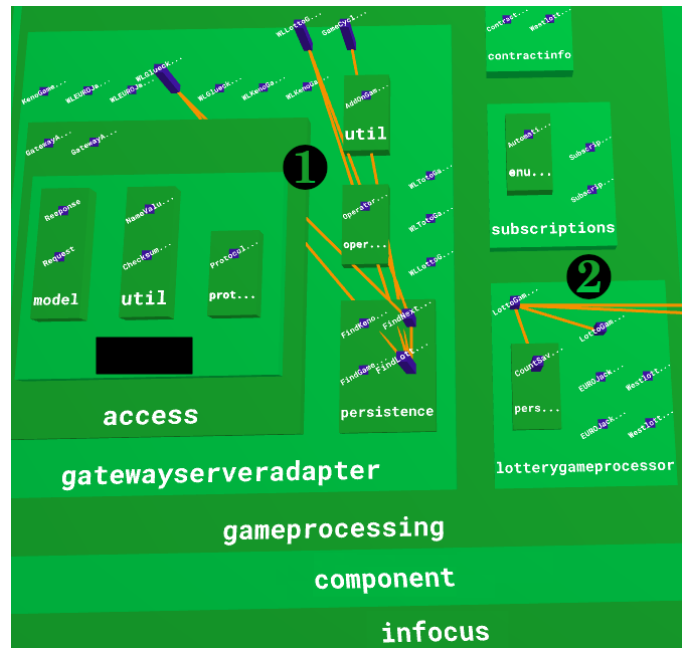


Fig. 8. Visualization of *in|FOCUS'* runtime behavior with *ExplorViz* when loading and filling out a lottery ticket as user.

a software system's runtime behavior to support the decomposition of a legacy, monolithic architecture into microservices. We combined established analysis techniques for microservice decomposition, such as the bounded context pattern of domain-driven design, and refined the architectural design via dynamic software visualization to identify appropriate microservice candidates. Especially employing runtime software visualiza-

tion using the live trace visualization tool *ExplorViz* proved to be a valuable step within our migration approach. Thus, we were able to confirm and refine the service boundaries which were obtained from the initial domain analysis.

We successfully applied our approach in collaboration with the German IT service provider *adesso SE* to their real-world, legacy lottery application *in|FOCUS* to identify potential microservice decompositions for their existing layered monolithic Enterprise Java system. The redefinition of the *Payment* service and the discovery of the additional *TicketManager* service showcase the benefits of dynamic analysis for microservice decomposition in Section V. By monitoring the runtime behavior of *in|FOCUS*, we gained new insights which were previously overlooked by the static analysis (see Section IV). The development team of *in|FOCUS* will use our results as foundation for a future decomposition into microservices.

After applying our approach to a real-world application for the first time, we now can derive certain lessons learned for other practitioners and researchers. Of course, the domain analysis is key for the overall approach. We incrementally refined intermediate and resulting data of the analysis with the help of documentation and discussions. Furthermore, we internally explained the domain to each other. This highlighted uncertainty that was then reinvestigated.

In the future, we plan to improve our approach and apply it to aid the migration process of additional software systems in collaboration with industrial partners.⁷ Specifically, we will investigate how we can consider design issues for microservices, e.g., code replication and non-functional requirements such as team size or security. Additionally, we enhance *ExplorViz* capabilities regarding filtering executed traces to better analyze specific sequences of communication within the observed software system. Moreover, we will extend its 3D software visualization towards collaborative Virtual Reality to offer an immersive user experience and natural interaction based on [25], [26].

REFERENCES

- [1] S. Newman, *Building microservices*. O'Reilly, 2015.
- [2] 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.
- [3] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating Towards Microservices: Migration and Architecture Smells," in *Proceedings of the 2nd International Workshop on Refactoring*, Montpellier, France: ACM, 2018, pp. 1–6. DOI: 10.1145/3242163.3242164.
- [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. DOI: 10.1109/ICSA.2018.00012.
- [5] Z. Dehghani, *How to break a monolith into microservices*, 2018. [Online]. Available: <https://martinfowler.com/articles/break-monolith-into-microservices.html>.
- [6] *Using domain analysis to model microservices*, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis>.
- [7] *Identifying microservice boundaries*, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries>.
- [8] J. Stenberg, *Strategies for decomposing a system into microservices*, 2018. [Online]. Available: <https://www.infoq.com/news/2018/06/decomposing-system-microservices/>.
- [9] E. Evans, *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley, 2004.
- [10] M. Fowler, *Ubiquitous language*, 2006. [Online]. Available: <https://martinfowler.com/bliki/UbiquitousLanguage.html>.
- [11] M. Gysel, L. Klbenner, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing*, M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, Eds., Cham: Springer International Publishing, 2016, pp. 185–200.
- [12] L. Baresi, M. Garriga, and A. De Renzis, "Microservices identification through interface analysis," in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds., Cham: Springer International Publishing, 2017, pp. 19–33.
- [13] H. Knoche and W. Hasselbring, "Using Microservices for Legacy Software Modernization," *IEEE Software*, vol. 35, no. 3, pp. 44–49, May 2018. DOI: 10.1109/MS.2018.2141035.
- [14] 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*, 2017, pp. 243–246. DOI: 10.1109/ICSAW.2017.11.
- [15] C. Fan and S. Ma, "Migrating monolithic mobile application to microservice architecture: An experiment report," in *Proceedings of the IEEE International Conference on AI Mobile Services (AIMS)*, Jun. 2017, pp. 109–112. DOI: 10.1109/AIMS.2017.23.
- [16] M. Weninger, L. Makor, and H. Mssenbck, "Memory leak visualization using evolving software cities," in *Proceedings Symposium on Software Performance (SSP 2019)*, Nov. 2019.
- [17] J. Waller, C. Wulf, F. Fittkau, P. Dhring, and W. Hasselbring, "SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency," in *1st IEEE International Working Conference on Software Visualization (VISOFT 2013)*, Sep. 2013, pp. 1–4. DOI: 10.1109/VISOFT.2013.6650520.
- [18] V. Khononov, *Bounded contexts are not microservices*, 2018. [Online]. Available: <https://vladikk.com/2018/01/21/bounded-contexts-vs-microservices/>.
- [19] 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: 10.1016/j.infsof.2016.07.004.
- [20] 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)*, AIS, May 2015, pp. 1–13. DOI: 10.18151/7217313.
- [21] 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 (VISOFT 2015)*, IEEE, Sep. 2015, pp. 36–45. DOI: 10.1109/VISOFT.2015.7332413.
- [22] 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)*, Mai 2015, pp. 266–276. [Online]. Available: <http://eprints.uni-kiel.de/28324/>.
- [23] C. Zirkelbach, A. Krause, and W. Hasselbring, "Modularization of research software for collaborative open source development," in *The Ninth International Conference on Advanced Collaborative Networks, Systems and Applications (COLLA 2019)*, Jun. 2019, pp. 1–7.
- [24] P. Sadalage and M. Fowler, *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*. Addison, 2012.
- [25] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring software cities in virtual reality," in *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISOFT 2015)*, IEEE, Sep. 2015, pp. 130–134. DOI: 10.1109/VISOFT.2015.7332423.
- [26] C. Zirkelbach, A. Krause, and W. Hasselbring, "Hands-On: Experiencing Software Architecture in Virtual Reality," Kiel University, Research

⁷During the migration process, we employed the tools *Structure101*, *ExplorViz*, and *DBeaver*. These tools are publicly available on the related websites. Corresponding documentation may help other researchers or practitioners to replicate our process in a similar setting.

Report, Jan. 2019. [Online]. Available: <http://eprints.uni-kiel.de/45728/>.