Migrating Monoliths to Microservices-based Customizable Multi-tenant Cloud-native Apps

Sindre Grønstøl Haugeland
University of Oslo
Oslo, Norway
sindrgro@ifi.uio.no

Phu H. Nguyen, Hui Song
SINTEF
Oslo, Norway
firstname.lastname@sintef.no

Franck Chauvel
Axbit
Molde, Norway
franck.chauvel@axbit.com

***Abstract*—It was common that software vendors sell licenses to their clients to use software products, such as Enterprise Resource Planning, which are deployed as a monolithic entity on clients' premises. Moreover, many clients, especially big organizations, often require software products to be customized for their specific needs before deployment on premises. While software vendors are trying to migrate their monolithic software products to Cloud-native Software-as-a-Service (SaaS), they face two big challenges that this paper aims at addressing: 1) How to migrate their exclusive monoliths to multi-tenant Cloud-native SaaS; and 2) How to enable tenant-specific customization for multi-tenant Cloud-native SaaS. This paper suggests an approach for migrating monoliths to microservice-based Cloud-native SaaS, providing customers with a flexible customization opportunity, while taking advantage of the economies of scale that the Cloud and multi-tenancy provide. Our approach shows not only the migration to microservices but also how to introduce the necessary infrastructure to support the new services and enable tenant-specific customization. We illustrate the application of our approach on migrating a reference application of Microsoft called SportStore.**

***Index Terms*—Microservices, Migration, Customization, Multi-tenancy, Cloud-native, SaaS**

I. Introduction

Migrating legacy software applications to become Cloud-based software applications, or even Cloud-based software-as-a-service (SaaS), is an undeniable trend. However, together with the great benefits of Cloud-based SaaS [1] come some big challenges that are still remaining to be addressed.

Monolithic applications have been the prevailing architecture for enterprise applications after the emergence of frameworks like J2EE around 2000. As a result of this, many legacy systems are performing and assisting in essential tasks in organizations. Large legacy enterprise solutions make use of monolithic architecture. Many companies today still have applications following monolithic architecture where all functions are coupled and built together as a single, interconnected unit. Monoliths have many drawbacks. They are large and complicated, making them difficult to change, add new features, or adopt new technologies. Their large sizes also make them slower to move around networks, start or restart on failure and also inhibits scaling with unclear resource requirements. Reliability is impacted as even simple bug fixes cause the entire application to be updated in every deployment.

Following the trend of cloud computing, enterprise software vendors are moving from single-tenant on-premises applications to multi-tenant (Cloud native) SaaS [2]. Customer companies no longer buy a license from the vendor and install software products in their own premises. Instead, they subscribe to an online service, which is also used by other customers, known as tenants of the service. The SaaS model brings new challenges to the software vendors with regard to enabling customization. It is not possible for any tenant to directly edit the source code of the same product service shared by other tenants. Software vendors must enhance the SaaS model with the ability to enable tenant-specific customization in the multi-tenant context.

In this paper we explore the current approaches used in the industry when migrating enterprise applications from a monolithic architecture to the microservices architecture and the different approaches used when transitioning an application from single to multi-tenant. Through reviewing the existing literature we found a number of different approaches that all accomplish one of these two goals, either focusing on the migration from one architecture type to the other or transitioning from single- to multi-tenant. Both microservices architecture and multi-tenancy offer additional benefits to the end-users of the application and the developers. Combining these two principles allows us to better utilize the economies of scale and the resource sharing found in SaaS applications. In the end, we aim to suggest an migration approach for cases where the target application follows a microservice architecture while also allowing tenants to customize the business logic to better fit their needs in a multi-tenant context.

Migrating to microservices architecture (MSA) is the right way forward for legacy systems to be modernized [3], [4]. There are huge benefits for migrating to MSA such as maintainability and scalability in the long run [5], e.g., by adopting DevOps and benefiting from Cloud-native elasticity [6]. Microservices can be packaged and deployed in isolation from the main product, which is an important requirement for multi-tenant context. Moreover, independent development and deployment of microservices ease the adoption of continuous integration and delivery, and reduce the time to market for each service. Independence also allows engineers to choose the technology that best suits one and only one service, while other services may use different programming languages, database,

This is the author accepted version of an article published in 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)
https://doi.org/10.1109/SEAA53835.2021.00030

etc. Each service can also be operated independently, including upgrades, scaling, etc.

The main contribution of this paper is the migration approach from monolith to MSA with specialised support for making the target MSA as customization-ready.

We present the result of our migration approach through a case-study describing each step of our approach while applying it to an existing application following the Model-View-Controller pattern. Our approach focuses on three stages during the migration, analyzing and breaking down the application into small bounded contexts, transforming the existing infrastructure to fit the new architecture and implementing functionality from the contexts as separate microservice, and finally adding the necessary components to support tenant-specific customization in the multi-tenancy context.

The remainder is structured as follows. Section II gives the definitions of the key aspects in this work. To provide a better understanding of the motivation and the requirements for our migration approach, Section III discusses a legacy e-commerce application that needs to be migrated. Then, we describe our migration approach in Section IV. An evaluation of our approach is presented in Section V. We discuss the related work in Section VI. Finally, Section VII concludes the paper.

II. Background

In this section we provide some background for the paper. We briefly introduce the main concepts for the migration.

A. Monolithic Applications

Monolithic application architecture is a common pattern that software applications follow. The pattern contains all the different layers of an application, including presentation, logic and persistence. All of which are contained within a single deployable package. The monolithic pattern is simple to deploy, scale and develop initially, but as the application grows and becomes more complex and developers encounter some new drawbacks with this pattern. Over time the single code-base for projects grows, and getting a complete understanding of all the internal complexities can become overwhelming. The frequency of changes in the application can potentially be an issue when the size of the application grows. Even the smallest changes require redeploying the entire application.

B. Microservice Applications

Applications following the microservices pattern consist of loosely coupled and highly specialized cohesive services, that work together to provide functionality. Compared to the monolithic architecture each service provides their own logic for a small part of the application. All these smaller services are then combined and hidden from the end-user behind a proxy, like an API-gateway.

C. Migration

When migrating software we split the process into three general phases [7]. The initial recovery of functionality, transformation of the current architecture and re-implementation following the target architecture. We briefly recall two popular migration approaches namely Strangler and Blueprint as follows.

1) Strangler: The Strangler approach is a typical approach coined by Fowler [8]. The whole premise of this strategy is to create new systems around the edges of the existing system. This approach includes different ways of diverting the calls to the old system, either by event interception, where the edge system taps into the message-stream intended for the original system and redirecting calls as new services are implemented. The alternative is to use asset capture, working with simple orders or specific customers.

2) Blueprint: The blueprint approach serves as a template for further adjustment, depending on the goals of the migration. The approach consists of two parallel tasks, building the required infrastructure to support the new system, and the actual migration. This approach uses aspects from domain-driven design (DDD) [9] to separate different functionality into bounded contexts. These contexts are then iteratively migrated into services or a set of services in the new architecture. Identifying these contexts is normally done by analyzing source-code, technical documentation, and in some cases, from interviews with developers that have worked on the pre-existing system [10]. The services migrated should ideally include everything except the UI, implementing the logic of the bounded context and a form of storage for the data related to the service.

In parallel to this process, the required infrastructure for the new architecture should be set up. While the existing infrastructure might be able to support a small number of services, future migration and expansion might require a more specialized infrastructure that better support the system.

D. Multi-Tenancy

A multi-tenant application serves multiple customers or tenants through an application shared by all the users [11]. Multi-tenancy is prevalent, particularly in cloud-hosted software. Since the application instance is shared among the different users, the software only solves a common set of problems for the users or a problem that the majority of the different users have [12]. Since the application is shared among multiple tenants, costs associated with the infrastructure and operations of the servers are also shared between the tenants, resulting in lower overhead for the application compared to running individual instances for each customer

III. A Motivational Example

In this section, we present a motivational example of why and what are the requirements for migrating a monolith to MSA that is customization-ready. We use a SportStore application as an example.

SportStore, which is a web-based store for sports equipment, is a software product of software vendor A. SportStore provides many of the essential features of an online shopping system such as user management, catalogue, shopping cart and checkout. It is implemented in .NET Core with Views,

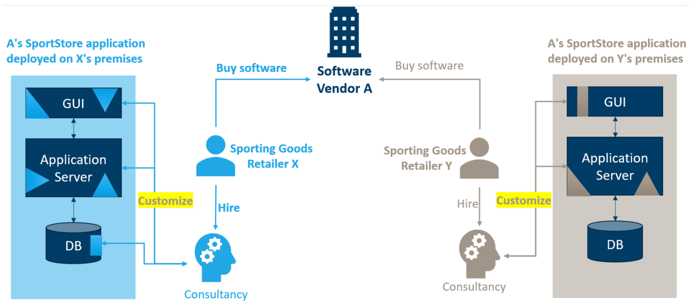
A's SportStore application deployed on X's premises
GUI
Application Server
DB
Customize
Buy software
Software Vendor A
Sporting Goods Retailer X
Hire
Consultancy
Buy software
A's SportStore application deployed on Y's premises
Sporting Goods Retailer Y
Hire
Customize
Consultancy
GUI
Application Server
DB

Fig. 1. Companies often need to customize software deployed for them

Models, and Controllers for ordering, product catalogs, and a session-based shopping cart. Software vendor A has sold and deployed separately their SportStore product for many sporting goods retailers, including two big ones: retailer X and retailer Y (Fig. 1). The big retailers such as retailer X and retailer Y often do not use the SportStore as-is but hire either software vendor A or a third-party consultant to customize or redevelop the SportStore product further according to their own specific needs. Retailers could have different business models leading to different requirements for customization.

Following the trend of cloud computing [2], software vendor A is migrating their software products such as SportStore to become multi-tenant (Cloud-based) Software as a Service (SaaS). Customer companies such as sporting goods retailers no longer buy a license from software vendor A and install it in their own premises. Instead, they subscribe to an online service, which is also used by other customers, known as *tenants* of the service. From each tenant's perspective, they still have the SportStore product as their own even though this SportStore-aaS is also used by other tenants. The SaaS model brings new challenges to software vendor A with regard to enabling customization, which is often required by big retailers like retailer X and retailer Y. It is not possible for any tenant to directly edit the source code of the same product service shared by other tenants. A major challenge is to ensure tenant-isolation while enabling tenant-specific customisation, which means that no customisation specific to a tenant shall ever affect any other tenants. What software vendor A must do to up their game for cloud computing model is finding a method to refactor and migrate their monolithic SportStore product to become **multi-tenant customizable (Cloud-based) SaaS**.

IV. Our Migration Approach

In this section, we give a brief overview of our approach (IV-A) and how it relates to multi-tenancy and the ability to provide deep customization for tenants (IV-B).

A. Overview of Our Approach

Our approach focuses on migrating applications that follow the MVC design pattern. It draws inspiration from the migration approach proposed by E. Wolff [13] in a survey about migration approaches, and the generic re-engineering tool in [7]. Moreover, we adopt the Blueprint approach to include the initial phase of the proposed tool by Kazman that allows the migrating party to gain an understanding of the current application before starting the migration. The approach becomes a three-phase approach, where the initial phase consists of analyzing the application and discovering bounded contexts and domains in the application according to domain-driven principles [9]. The next two phases occur in parallel, where we extract functionality from the existing application while building the necessary infrastructure to support the new services. Note that in this paper we do not address the constraints of migrating currently in-use applications, which the Strangler approach [8] can do best. We rather focus on how to logically migrate a monolithic application to become customizable in a multi-tenant context. Our approach can be adopted to be part of the Strangler approach for migrating currently in-use applications.

Three different phases of our approach: the analysis, transformation and implementation. Each of these different phases focuses on a separate aspect of the migration. During the initial phase, we analyse the application we are migrating in order to determine how the internals of the application works, and how we can split up the different modules in the application into separate microserivces. The goal is to identify and group these modules into domains or contexts that focus on a specific areas of the application.

The second phase of the migration consist of transforming the existing infrastructure of the application to fit the new architecture target. If the existing infrastructure can not be transformed, or if more effort needed to transform the infrastructure, we implement additional infrastructure to support the new application architecture (Fig. 2). Additional architecture that we need in order to support the MSA compared to the monolithic architecture is isolated storage for the different services, a gateway that connects external clients, like a web-application or external third party applications, to the microservices, and a back-end communication system (e.g., an event bus for enabling event-based customization [14]). The final phase consists of implementing the services we have identified during the initial phase, and connecting them to the infrastructure we add during the second phase (Fig. 2).

B. Multi-tenancy and Deep Customization

Our approach aims at enabling the target architecture to be customizable for multi-tenant context as presented in [15], [16]. The approaches in [15]–[18] offer tenants a way to (deeply) customize the functionality of the multi-tenant application without interfering with behavior for other tenants. The customization-driven aspect makes our approach different from other migration approaches. Adding customization support for tenants can be done using the tenant-manager as a lookup table (Fig. 2). Tenants register their customizations with the tenant manager. These can be either standalone services outside the main application, exposing endpoints that the service can redirect calls to. The services and the customized functions that they provide need to adhere to a predetermined stable interface defined by the developers of the main applica-

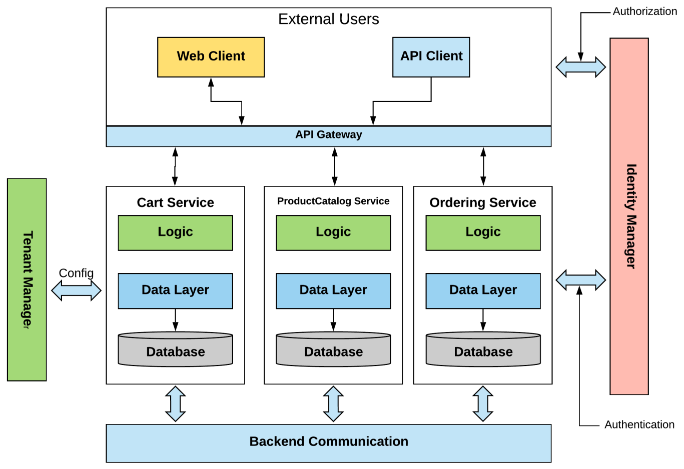

Fig. 2. Target Architecture

tion. This interface serves as a contract between the service and the customized endpoint, describing the expected result that the service needs to continue operations after the customized function has been called.

First, we focus on introducing multi-tenancy to the application. To support multi-tenancy, we need a system for Identity Access Management (IAM), and to support customization of the application for the tenants and to configure the storage to isolate tenant data, we need a tenant manager. The tenant manager provides all the registered customizations and endpoints for the logged-in user, which is retrieved using a bearer token issued by the IAM system. Tenant isolation at application level is crucial to avoid data leaks problem between tenants as raised in [19], and more aligned with the concept security by design [20], [21]. With the tenant manager and the IAM system in place, we start adding support for customization. We use the tenant manager to return external endpoints to customized functionality in cooperation with the IAM system to ascertain the "tenantID" of the user. The main service then reroutes the request to the external endpoint along with the information required by the customized function.

To support the customization of the services, they need to use both the identity manager and tenant manager. The tenant manager keeps a record of all the customizations associated with a specific tenant, which can be looked up by services after querying the identity server for the user profile of the token attached to the request. We have two different scenarios for how the tenant manager is used by the services. One scenario is where the tenant has registered customization for some of the functionality, and another where the tenant uses the default functionality implemented in the service already. For both of these scenarios, the configurations are retrieved from the tenant-manager by the service. The response is cached for quick access and reduced network traffic. The tenant manager then push updates to the services when configurations are updated.

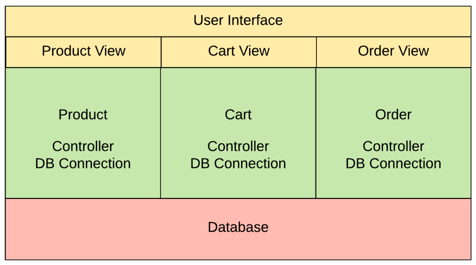

Fig. 3. The monolithic application

V. Evaluation

To validate the approach, we applied it to the SportStore application [22], whose monolithic architecture is simplified in Fig. 3. The application following the MVC pattern [22] is a web-based store for sports equipment. We identify the different groupings in the application during the first phase of the migration. The sport store application consist of three different groups; products, carts and orders. Each of these groupings has their own models, views and controllers. The functionality in these groups are closely related to each other. After the initial analysis we introduce the additional infrastructure needed to support the new architecture. In our case we add an API gateway that connects the user interface to the services through a single entry point, and a message-broker that the services use to communicate. We then implement the groupings we identified during the first phase as separate micro-services with isolated storage. Once we have extracted the services from the old application we connect them to multi-tenant specific infrastructure.

SportStore is implemented in .NET Core with Views, Models, and Controllers for ordering, product catalogs, and a session-based shopping cart. We use these "groupings" as our bounded contexts during the analysis and extract functionality during the decomposition part of the migration. After this, we start implementing services to cover the functionality of the existing application and set up the infrastructure to support it (Fig. 4). The infrastructure includes typical components like the API-gateway and a form of back-end communication for the services. Fig. 5 shows the target architecture of the SportStore application that we have used for our migration experiments. We present the details of our migration process in the following subsections.

The MVC pattern offers a natural separation between the different layers. During the analysis and implementation we extract or replicate from the controllers in their own microservices. The application still has controllers; however, these only serve as a way to make calls to the services. In a way, they hide the fact that the back-end of the system is spread out into microservices. The goal is to first analyze and break down the modules in the sportsStore application into separate bounded contexts following Domain-driven design and implementing these contexts as microservices.

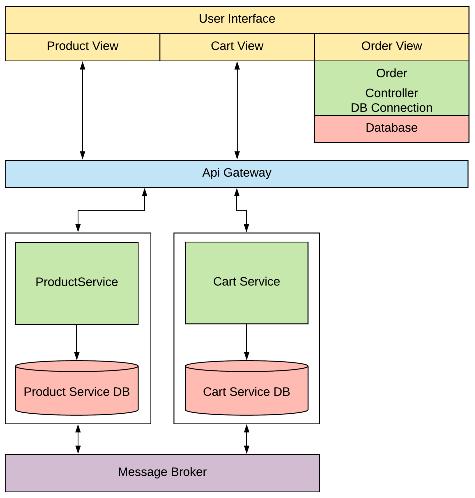

Fig. 4. Migrating functions to microservices with API gateway and Message Broker

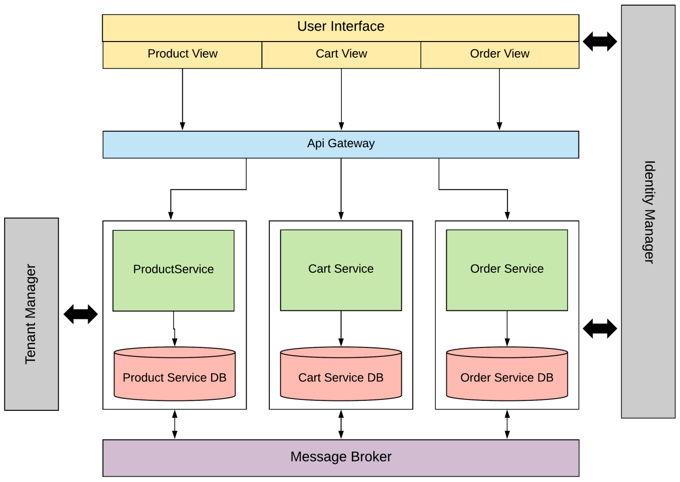

Fig. 5. Target architecture with customization possibility for multi-tenant

A. Analysis and decomposition

Once the structure and architecture of the existing application are analyzed and mapped out, we start decomposing the application into separate domains. During the analysis, we found three different domains within the application—the product domain, containing a template for the products. The product template consists of a productID, product name, a short description of the product, the price, and the category of the product. The application stores the model in an MSSQL database, which is stored in an Entity Framework repository, with supporting methods for retrieving, updating, adding, and deleting products in the repository. The shared resource between the product domain and the cart is the product (Fig. 4). Each of these domains needs to have a shared understanding of what a product is. The cart domain contains the cart for the current session, and it consists of a list of cartlines, as well as methods for adding and removing items to the cart. Additionally, there are methods for computing the total cost of all the items in the cart and clearing the cart. The cartline class represents an item that has been added to the cart. It is made up of a productId, quantity, and the id for that specific cartline. The cartline resource is shared with the third domain, the orders-domain (Fig. 5).

B. Additional Infrastructure

The migration to microservices requires some additional supporting infrastructure. The introduction of an API gateway and back-end communication is an integral part of the second phase of the migration.

1) API Gateway: The API gateway is an integral part of the microservice architecture. The gateway serves as a connecting layer for clients and other consumers of the services, rerouting requests to the right microservice, serving as a proxy for the different services.

2) Message Broker/Event Bus: In the prototype, the back-end communication moved through different stages. Each of these stages aimed to further decouple the microservices from each other. The first iteration of the back-end communication implemented direct synchronous calls from service to service. The calls are made to endpoints that the services expose to each other. The second iteration involves adding a message broker to the application to decouple the services further and adding an asynchronous way for them to orchestrate events affecting multiple services. The message broker is implemented using RabbitMQ with a topic-based broker. Messages are tagged with a specific topic, for instance, orderCreate, when the cart of a session is checked out in the sportStore. The new message is then routed to all the queues matching the orderCreate tag, and the productOrder service then consumes the messages and creates the order in a FIFO order. The topic message broker was initially chosen to facilitate customization for the different services further down the line, where messages of a specific tenant would be published to a queue being consumed by that tenant's customized microservice.

3) Identity Server: In this section, we look at how the Identity server component of the application works. We describe how the different features provided by IdentityServer and OpenId connect help with authentication and authorization for the different tenants. Clients represent the applications that can request tokens from the identity server. In our case, only the web store of the sportStore application use the tokens on behalf of the user. The grant types we define, specify how the clients can interact with the Identity Server. The tokens issued allow both services and users to interact directly with the identity server, because of the grant types we use. If we were to define individual grants for the service and users, we would use the Client Credentials type for the service, and OpenID grant type for the users to interact with the server. The identity resources define the functionality enabled by the identity server. The OpenID identity resource allows users to log in via the OpenIDConnect login screen, while the profile resource type allows the services to retrieve the claims of the users to check for customizations later on. The API

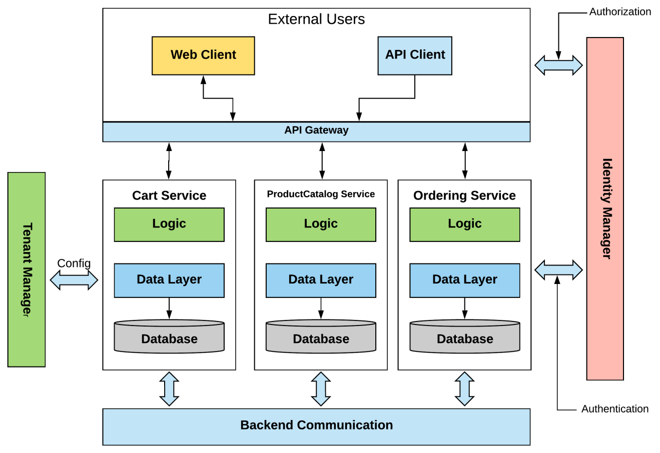

Fig. 6. Final architecture of the application after migrating all the different services, and implementing the different additional infrastructure components needed for the the new architecture

resource allows the client to access the gateway by defining and associating access to a specific scope. In our case, we only need one resource since all the services are hidden behind the API gateway, and access to all of them on the user's behalf is necessary to provide the full functionality of the system.

4) Tenant Manager: The tenant manager is an essential component of the multi-tenant aspect of the application. We use it to configure the persistency layer for the tenants and as a lookup for customized endpoints. Using the "/userInfo" endpoint of the identity server, the tenant manager can retrieve all the claims belonging to the logged-in user. The claims contain information about where the user belongs and what right he or she has for the services. With the "tenantID", we can look up the customization for the called functionality on the main service. Initially, we implement the tenant manager as a mock service in the main service. The TM uses the token from the initial request and sends a request to the "/userinfo" endpoint of the identity server, which returns all the claims associated with the token.

C. Implementation

In this section, we go through the process following our migration approach and applying it to the migration of the SportsStore application, first to the microservice architecture, and then implementing multi-tenancy for the application. We split the migration up into different phases. Each phase includes the extraction of a single service from the pre-existing system, as well as adding the necessary infrastructure to support the new migrated functionality from the monolith.

In Figure 6 we see the final architecture of the application after all the different services and infrastructure components have been implemented or migrated.

1) Migration: We split the migration itself into different phases, related functionality from the existing application during each of these phases and adding the necessary infrastructure to the new application needed to support the extracted components from the pre-existing application.

The initial phase of the migration consists of the analysis and reverse engineering of the pre-existing application. During phase one, the application is still in a single monolithic piece. At this stage, the application consists of three different layers typically found in MVC applications. A user-interface that represents the view, controllers that contain the application logic, and a persistent storage layer that handles the storage of the models in different databases.

The second phase of the migration starts by picking a service or some functionality for migration. Ideally, this functionality should already be loosely coupled to the rest of the code in the monolithic application to limit any dependency back to the monolith. For this phase, we chose to focus on the product module of the SportsStore application. The product module contains all the logic associated with displaying products from the database, adding and updating products in the database, and adding or removing products to a customer cart. We add the additional infrastructure pieces associated with a microservice architecture before we migrate the service. The API-Gateway forwards and redirects calls originating from the web applications to the specific services and act as a unifying endpoint for the View instead of having the calls directed to the specific service it targets an endpoint at the gateway.

In Phase 3 of the migration, we extract another service from the monolith. With two services extracted, we need a way to orchestrate how they cooperate. We introduce the message broker as an additional piece of infrastructure to help with orchestration. The message broker keeps a queue of messages published by all services. Services can then subscribe to the message queue to consume the messages and perform actions with the content of the message. The new service is added to the API-gateway behind the downstream endpoint "/cart". Collaboration and orchestration between the cart service and product service use the message broker to add and remove products to the customer cart. The primary use of the broker at this time is to get information about the products in the cart. Using the session data, we publish a message to the broker requesting a lookup in the products database for the items in the cart. Once the product service retrieves the message from the queue, it aggregates all the products from the cart into a list before returning it to the cart service

After the fourth phase of the migration, the application is now following the microservice architecture design pattern. The functionality from the monolith has moved into individual services decoupled from each other. All calls from the web client are passed through the gateway and forwarded to the appropriate service. Orchestration and communication between the services happen through the use of a message broker. The development and deployment of different services are isolated from each of the other services. There is also a clear separation of the different layers of the application allowing tenants to customize the different services.

The final phase of the migration introduces more infrastructure to support multi-tenancy. We add an IdentityManager to support login, authentication, and authorization of users and a TenantManager to provide the services with endpoints for tenant-specific customizations. To support the customization of the services, they need to use both the identity manager and tenant manager. The tenant manager keeps a record of all the customizations associated with a specific tenant, which can be looked up by services after querying the identity server for the user profile of the token attached to the request. We have two different scenarios for how the tenant manager is used by the services. One scenario is where the tenant has registered customization for some of the functionality, and another where the tenant uses the default functionality implemented in the service already.

2) Tenant Customization: Configurations are retrieved from the tenant-manager by the service. The response is cached for quick access and reduced network traffic. The tenant manager then push updates to the services when configurations are updated. Once the service has the information about tenant customization, it reroutes the information to the address and endpoint specified. The customized endpoint then performs the function and prepares the result for the response back to the ordering service. Before the response is returned to the ordering service, it needs to be fitted to the interface determined by the provider of the application. The interface includes the data types that should be present in the response, how the data should be structured. Necessary data types include, for example, the total cost of the order, and how the object that the customized service should be structured.

One of the main benefits of the microservice architecture is the decoupled nature of the individual services. Compared to a monolith state in which the pre-existing application was initially, any changes to to code required a full redeployment of the system. The ease of development due to the isolated nature of the services will be used to determine the benefit of the migration. Time from code completion to live is measured from the feature or change is implemented until the change has propagated to the actual application and is visible or observable to the end-user. Affected components is also a metric used to determine the benefits of the migration. In the monolithic system, a change would affect all the different modules through the redeployment of the application. Once more services are extracted we expect that the number of affected components/modules that has to be redeployed as a result of a change to decrease. To measure this we use the different contexts or domains discovered during the initial analysis. A component is considered affected when it has to be redeployed due to a change somewhere else in the system. While the number of affected components should decrease we also expect that there will be changes affecting other services.

We break the categorisation of these effects into the different phases during the migration. We measure the effect of a change during each phase, and in all the existing components that we have either belonging to the monolith or as separate services. The additional infrastructure of the application once we start the migration are considered as separate components for this metric.

VI. Related Work

This section discusses related approaches for migrating from single-tenant to multi-tenant, and monoliths to MSA.

A. Single-tenant to Multi-tenant

One of the primary challenges with multi-tenant applications, according to Kwok et al. [11] is that the application has to deliver a shared product to multiple tenants, resulting in one-size-fits-all solutions even though the different user-groups might have slightly different needs from the application.

A way to solve this is by allowing the individual users to customize different aspects of the application for their needs, but this introduces additional challenges, according to Walraven et al. [12]. Ad-hoc handling of changes related to one specific tenant can potentially affect all the tenants of the application Multi-tenant applications is another way to take advantage of economies of scale. Multi-tenant applications share resources between multiple different user groups or tenants, keeping the tenant-specific data separate.

Migrating an application from single to multi-tenant is a large undertaking. There are certain requirements that need to be in place before the application can be migrated, Furda et al. [19] describe an approach for migrating legacy single-tenant applications to multi-tenant. The approach moves through three different phases during the migration. In the initial phase, the legacy system supports only a single tenant, and its structure and architecture have not been changed yet. The second phase focuses on changing the design pattern of the application into one that is better suited for multi-tenancy.

In [23], the authors propose a lightweight reengineering approach to migrate a single-tenant software system into a multi-tenant one. The targeted multi-tenant software system can provide capabilities for tenant-specific layout styles, configuration and data management. Our migration approach enables tenant-specific customization, which is beyond configuration capabilities.

B. Migrating to microservices

The customization-driven aspect makes our approach different from other migration approaches like [3], [4], [24]. In [3], the authors present their migration approach using migration patterns for managing service decomposition and data isolation and replication. While in [4], the authors present an incremental approach of re-engineering a mission critical banking system that led to reduced complexity, lower coupling, higher cohesion, and a simplified integration. Each of the approaches has its specific contexts where they are the most suitable. These approaches focus on migrating live systems or systems that have been used extensively by organizations. For our prototype, we found the blueprint approach most suitable due to the ”stale” state of the application. By stale, we mean that the application is no longer actively developed. All the approaches we found follow a similar pattern, made up of three-phases: Reverse-engineering, transformation, and implementation. What separates them is the focus they put on transforming and moving the existing functionality into new services. The customization-driven aspect makes our approach different from other migration approaches. The primary contender for the approach we chose was the Strangler approach. Comparing this approach to the blueprint approach, we see that the focus is on following the new architecture with any new functionality. Because the amount of new functionality we are adding to the system is limited, we landed on the blueprint approach for our MSA migration.

VII. Conclusions

This paper has presented an approach for migrating monoliths to microservices-based customizable Cloud-native SaaS. Our approach splits the migration into three stages, where we first analyze and break down the application into bounded contexts separating the different responsibilities and application areas. After the analysis, we start transforming the infrastructure to fit the MSA. This includes migrating information from databases related to the contexts discussed above, and setting up additional components necessary to support the new services, like the API gateway and the message exchange. Finally, we implement the functionality from the contexts as separate services and connect them to infrastructure. Once the functionality of the application has been migrated, we add the infrastructure necessary for multi-tenancy. After having the application in a suitable state, we can apply the approach to migrate it. The initial phase analyzing the application creates the foundation for the entire migration. It is the most crucial part of getting the migration right.

When it comes to the multi-tenancy transition, there are no other approaches that enables tenant-specific customization capability in a multi-tenant context. Our focus for this transition was laying the groundwork for customization further down the line. The primary concerns with multi-tenancy are avoiding noisy neighbors and ensuring that the tenant data is sufficiently isolated. Moving the customization outside the same execution context of the main product solves this. Customization no longer compete for computing resources with the main application, and the data of other tenants remain entirely isolated from the customization code.

VIII. Acknowledgments

The research leading to these results has partially received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement No. 958363 (DAT4.Zero), and from the Research Council of Norway under the grant agreement number 309700 (FLEET).

References

[1] A. Lenart, "Erp in the cloud – benefits and challenges," in *Research in Systems Analysis and Design: Models and Methods*, S. Wrycza, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 39–50.

[2] IDG, "2018 Cloud Computing Survey," Tech. Rep., 04 2018. [Online]. Available: https://www.idg.com/tools-for-marketers/2018-cloud-computing-survey/

[3] A. Henry and Y. Ridene, *Migrating to Microservices*. Cham: Springer International Publishing, 2020, pp. 45–72.

[4] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar, "Microservices: Migration of a mission critical system," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.

[5] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[6] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[7] R. Kazman, S. G. Woods, and S. J. Carriere, "Requirements for integrating software architecture and reengineering models: Corum ii," in *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, 1998, pp. 154–163.

[8] M. Fowler. (2004) Strangler Fig Application. [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html

[9] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[10] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29–2909.

[11] T. Kwok, T. Nguyen, and L. Lam, "A Software as a Service with Multi-tenancy Support for an Electronic Contract Management Application," in *2008 IEEE International Conference on Services Computing*, vol. 2, Jul. 2008, pp. 179–186.

[12] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen, "Efficient customization of multi-tenant Software-as-a-Service applications with service lines," *Journal of Systems and Software*, vol. 91, pp. 48–62, May 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121214000326

[13] E. Wolff, "Migrating monoliths to microservices: A survey of approaches," in *International Conference on Microservices 2019*, 2019.

[14] E. T. Nordli, P. H. Nguyen, F. Chauvel, and H. Song, "Event-based customization of multi-tenant saas using microservices," in *Coordination Models and Languages*, S. Bliudze and L. Bocchi, Eds. Cham: Springer International Publishing, 2020, pp. 171–180.

[15] P. H. Nguyen, H. Song, F. Chauvel, R. Muller, S. Boyar, and E. Levin, "Using microservices for non-intrusive customization of multi-tenant saas," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, New York, NY, USA, 2019, p. 905–915. [Online]. Available: https://doi.org/10.1145/3338906.3340452

[16] H. Song, P. H. Nguyen, and F. Chauvel, "Using microservices to customize multi-tenant saas: From intrusive to non-intrusive," in *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*, 2020.

[17] H. Song, P. H. Nguyen, F. Chauvel, J. Glattetre, and T. Schjerpen, "Customizing multi-tenant saas by microservices: A reference architecture," in *2019 IEEE International Conference on Web Services (ICWS)*, 2019, pp. 446–448.

[18] H. Song, F. Chauvel, and P. H. Nguyen, *Using Microservices to Customize Multi-tenant Software-as-a-Service*. Cham: Springer International Publishing, 2020, pp. 299–331.

[19] A. Furda, C. Fidge, A. Barros, and O. Zimmermann, "Chapter 13 - reengineering data-centric information systems for the cloud – a method and architectural patterns promoting multitenancy," in *Software Architecture for Big Data and the Cloud*, I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, and B. Maxim, Eds. Boston: Morgan Kaufmann, 2017, pp. 227–251. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128054673000132

[20] P. H. Nguyen, M. Kramer, J. Klein, and Y. L. Traon, "An extensive systematic review on the model-driven development of secure systems," *Information and Software Technology*, vol. 68, pp. 62–81, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001482

[21] L. Lúcio, Q. Zhang, P. H. Nguyen, M. Amrani, J. Klein, H. Vangheluwe, and Y. L. Traon, "Chapter 3 - advances in model-driven security," in *Advances in Computers*, ser. Advances in Computers, A. Memon, Ed. Elsevier, 2014, vol. 93, pp. 103–152. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128001622000038

[22] A. Freeman, *Pro Asp. net Core Mvc*. Apress, 2016.

[23] C. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. . Hart, "Enabling multi-tenancy: An industrial experience report," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–8.

[24] D. Taibi and K. Systä, "From monolithic systems to microservices: a decomposition framework based on process mining," in *8th International Conference on Cloud Computing and Services Science, CLOSER*, 2019.