Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

HANNES FLEISCHER

BACHELOR THESIS

APPLYING EVENT-DRIVEN ARCHITECTURE IN THE JVALUE ODS

Submitted on 13 July 2020

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 13 July 2020

License

Erlangen, 13 July 2020

Abstract

In a world of Big Data and data transparency “open data” might not be considered as an unknown term anymore. As the name suggests, it describes data that is commonly available and free to use for everyone.

The vision of the software “JValue Open Data Software (ODS)” is to provide a solid solution for the community to achieve data availability and homogeneity in representation regarding open data. With the provision of an user interface to configure the way open data is extracted and how it is presented, ETL (Extract-Transform-Load) processes are exceedingly facilitated.

The challenges of scaling large for a platform, based on a software orientated architecture, demand a solution that can handle a large set of interactions. The purpose of this thesis is to apply an “Event-Driven Architecture” to the already existing model of the software in order to fulfil these requirements. This is done by reevaluating the current software design that is composed of distributed Microservices and identify events of its current architecture with a technique called “Event-Storming”. The results will be applied to an architecture design that will be implemented in the “JValue ODS”.

Contents

Acronyms

AMQP Advanced Message Queuing Protocol. 35

API Application Programming Interface. 11, 13–15, 25, 34, 38, 49–52, 54

CI Continuous Integration. 8, 9

CQRS Command-Query Responsibility Segregation. 25, 49

CRUD Create-Request-Update-Delete Operations. 39, 45, 51

CUD Create-Update-Delete Operations. 25, 38, 39, 50

DDD Domain Driven Design. 3, 4, 29, 44

DDL Data Definition Language. 49

DevOPS Development Operations. 8, 9

DML Data Modeling Language. 50

EDA Event-Driven Architecture. 2, 20, 40

EERM Enhanced Relationship-Entity Model. 37

ODS Open Data Service. 27, 32, 35–37, 41, 43, 47, 48, 53, 54, 62

REST Representational State Transfer. 11, 13–16, 35, 38–41, 49–52

RPC Remote Proceedure Calls. 11

SJON JavaScript Object Notation. 14, 15, 45, 46

1 Introduction

Nowadays more and more importance gets attached to *data transparancy* and *knowledge exchange*, resulting in a demand for provisioning of data for general welfare purposes. Therefore an increasing amount of establishments is providing *open data* to satisfy these needs. *Open data* describes data, that is commonly available and free to use for everyone and is the main subject of *JValue Open Data Service* .
This software supports open data consumers when it comes to extracting and maintaining this data from various web services. It provides a platform for the general public to gain unified control over the ETL-Processes (Extract-Transform-Load) regarding the obtainment of open data from web interfaces, such as RESTful Application Programming Interfaces (APIs).

Extract - To accomplish higher availability this service not only stores already aquired data sets, it also enables the operators to choose a frequency to update the persistent data with data from related sources, giving the opportunity to respond to quota restrictions and to counteract scheduled downtimes.

Transform - Furthermore it reduces the efforts of handling various data representations, such as XML or JSON, from several sources by giving the user the opportunity to choose the structure of the data to be received. As a consequence a consuming application no longer has to take care of covering multiple formats when aquiring data from different sources, in order to achieve a specific goal.

Load - While many of these applications only use a specific share of the result sets from web service calls, they might suffer from efficiency losses, due to having to deal with data, they do not utilize. A solution for this problem is given by *JValue ODS* by serving a built-in and easy-to-use filter mechanism. It enables users to apply filters, in order to retain or omit data sections within the responses of open data webservices. This feature not only simplifies the processing of the data in applications, it enhances performance by decreasing webservice response timings when using *JValue ODS* as intermediate platform, resulting in an increase of “reponsiveness“.

Its structure is composed of different *Microservices*, whereas every of these components serves a specific purpose, contributing to the core functionality of the whole application.
With its architecture design *JValue ODS* follows the trend of modularizing “old-fashioned“ monolithic software structures to a coherent set of services, that represent specific all functional aspects the application. Many design patterns have emerged from this proceeding with the intention to provide a more beneficial solution, in terms of exchangability, scalability and robustness.
One particular pattern, this thesis is paying attention to, is the *Event-Driven Architecture* (EDA) in the context of *JValue ODS*. As *Microservices* describes the segmentation of software to isolated services, EDA is more concerned about the communication between these.

The current design of the software, regarding the inter-service communication, led to the consideration of a more flexible and scalable message system solution. The goal this thesis is trying to achieve is to move from the current orchestration or respectively communication mechanism of *JValue ODS* to a event driven choreography. This is done by analysing the current architecture and communication patterns with a technique called *Event-Storming Workshop*. The results of this workshop can be derived to establish a new architecture that encloses event-based messaging and serves as the basis for the implementation. It will furthermore investigate critical paths in the messaging flow, in order to substitude the communication patterns efficiently. The results of the implementation will be evaluated by executing stress tests, regarding the communication, to the former and new architecture and putting them into comparison.

2 Fundamentals

2.1 Domain Driven Design

As the Structure of the *Jvalue Open Data Service* is, to some extent, created by *Domain Driven Design* and the technique to identify events of its current architecture (*Event-Storming*) refers to it, a definition of this design pattern is required.

2.1.1 Definition

Domain Driven Design (DDD) is a technique for designing complex software by closing the knowledge gap between software developers and domain experts, such as project owners, business analysts, stakeholders, etc. in order to build meaningful models, that reflect business logic. It was first proposed by Eric Evans in his work "*Domain-Driven Design: Tackling Complexity in the Heart of Software*" and aims at providing more transparency to all participants of a software project, as well as designing flexible and robust software architectures. It can furthermore be understood as a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains [Eva04]. This modeling approach helps explaining complex relations on an abstract level by determining delimited boundaries and facilitates the design of an evolving software architecture that is a projection of core business concepts.

2.1.2 Domain and Bounded Context

A domain is, per definition, a "sphere of knowledge" and should be the primary focus for most software projects, when it comes to reducing complexity, according to Evans [Eva04]. In the context of an software application, domains can be determined by problems and objectives that have to be solved. A domain can furthermore divided into *Subdomains* that represent areas of capability, define business processes and represent functionality of a system [MT15].
Subdomains can moreover differentiated by three categories.

Core Domain - DDD mainly focuses on *Core Domains*, which are part of the business domain that represent the main purpose of the system. Primary importance is attached to this Subdomain due to having significant impact on the success of an organization [Ver13, p. 71].

Supporting Subdomain - These domains are essential for the business, but do not represent he core business logic. They rather have a supporting function for the Core Domain.

Generic Subdomain - A *Generic Subdomain* captures nothing special to the business, yet is required for the overall business solution [Ver13, p. 71].

Generally business models and the feasibility of technical implementation of these may differ and may be modeled in another way. *Bounded Contexts* represent coherent functional aspects of the system that might not fit exactly into a subdomain and define the range of applicability of each model [Eva04, p. 239]. They should be clearly separated from other contexts to ensure clear competences and naming conventions and may be connected to each other. In Addition a *Context Map* can be defined to provide a global overview of the contexts and relationships between them [Eva04, p. 239].

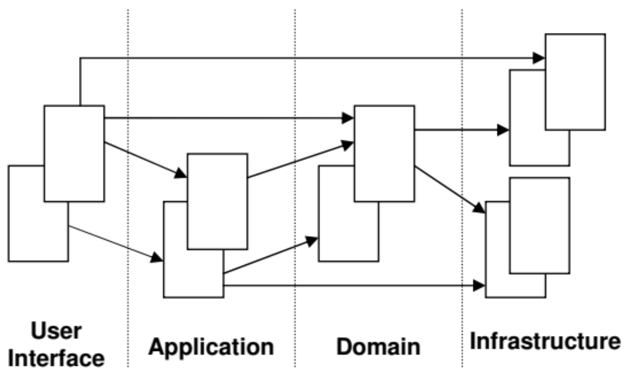
2.1.3 Ubiquitous language

When it comes to building a domain model, several parties may contribute to its design. When domain experts and developers collaborate on this model, the use of common language plays an important role. Domain experts may have a limited understanding of technical jargon of software development, but they use the jargon of their field. On the other hand developers may make use of descriptive and functional terms, regarding the discussion of systems [Eva04]. Therefore the use of an ubiquitous language is highly recommended to facilitate the communication between corresponding responsibles.

2.1.4 Layered Architecture

To avoid confusion of domain concepts with software technologies, decoupling of domain objects from other functions of the system might be mandatory. Layering of the architecture facilitates this process. The essential principle is that every element of a layer only depends on another element within the same layer or on layers "beneath". This specialisation allows a more cohesive design of each aspect and makes the design easier to interpret [Eva04, p. 63].
Typical Layers that can be implemented are shown in figure 2.6.

Figure 2.1: Layered Architecture [Avr07]



User Interface - The User Interface or Presentation Layer is responsible for showing information to the user and interpreting user commands [Eva04, p. 64].

Application Layer - the Application Layer is a thin layer which coordinates the application activity and does not contain business logic or rules. It rather coordinates tasks and delegates work to collaboration of domain objects to the *Domain Layer* [Eva04, p. 64].

Domain Layer - As the name suggests, this layer contains information about the domain. It further more delegates the persistence of business objects and their state to the infrastructure layer [Avr07, p. 38].

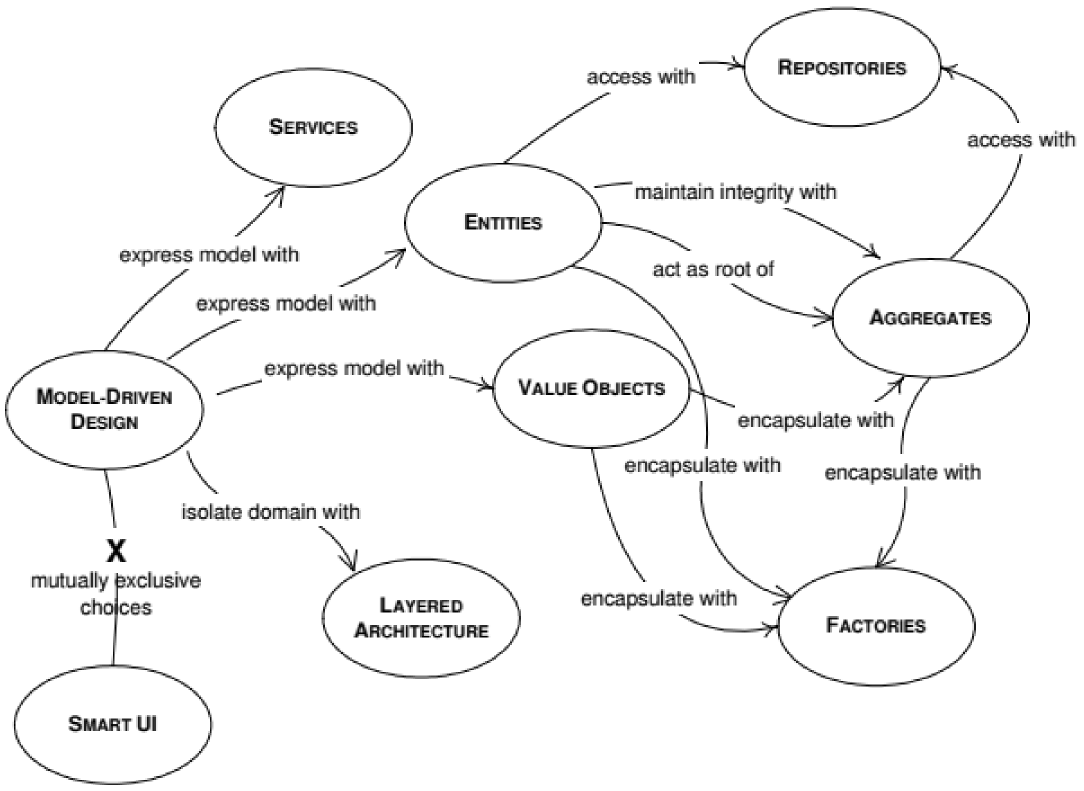
Infrastructure Layer - The Infrastructure Layer provides technical capabilities that support the other layers. It for instance assists on persisting domain objects, drawing widgets for the UI and message sending for the application [Eva04, p. 64].

2.1.5 Model-Driven Design

Whereas the domain model only covers resolving problems regarding the domain, the model driven design moreover focuses on the implementation on a lower level. The building blocks, as defined by Evans, stand for the elements of this model to be designed for the domain driven approach.

Entity Entities are elements, used to express model-driven design models. An *Entity* is an object with an individual identity, which remains throughout the software. It can exist with the lifespan of a system or extend it by being stored into databases [Avr07, p. 39].

Figure 2.2: The Building Blocks Of A Model-Driven Design [Avr07]



Value Objects *Value Objects* do not have their own identity and are purely describing domain-relevant attributes of entities, usually in the form of some quantity [MT15]. They have no state throughout the system might be moreover considered as an state of an *Entity*. They are therefore immutable objects and can be distributed among the system.

Aggregates Aggregates are composite domain objects and decrease complexity of a model by providing one interface for its coherent Entities and Value Objects. The root is an Entity and is the only object accessible from outside and holds references to any of the aggregate objects. With the provision of only one interface for those coherent domain objects, it ensures data integrity and enforces invariants [Avr07, p. 44].

Services Services represent domain concepts and are stateless [MT15, p. 435]. They are used when the behaviour does not conform with the concept of the Model-Driven Design using Entities and Value Objects. The main intention is to formulate business rules.

Repositories Repositories are used to manage the persistence and retrieval of aggregates usually using a database. It ensures the separation of the domain model and the data model by mediating between these two models [MT15, p. 525].

Factories The creation of domain objects by encapsulation. Factories can reconstitute object from a persistence model or create new domain objects , encapsulating complex domain logic [MT15, p. 470].

2.2 microservices

2.2.1 Definition

Since the architecture of *JValue ODS* is based on microservices, further explanation might be mandatory. Although there is no common definition of microservices, the segmentation or modularisation of software by functionality might be the best fitting term. A comprehensive definition was given by Nadareishvili et al. [Nad+16, p. 6]:

> “A microservice is an independently deployable component of bounded scope that sup- ports interoperability through message-based communication. microservice architec- ture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices.“

Furthermore a good approach of defining a microservice might be to describe its core functionalities and benefits in comparison with software monoliths.
While traditional software monoliths describe software, that keeps the whole functionality of a program at one place, the microservice pattern is more focused on separating the program into components, that represent all functional aspects of the former structure. The static architecture of the monolith leads generally to difficulties in terms of changeability, exchangability and team collaboration, whereas the microservice patterns seem to solve these problems. These advantages over the monolith led to a trend of splitting it into autonomous services to achieve a more agile development of software.

Referring to Newman [New15] the main key aspect is the autonomy of the services. Each microservice of an application is responsible for a specific purpose and does not interfere with other components of the infrastructure. Thereby sharing information between the services have to be accomplished vie network communication, such as requests over application programming interfaces (API) or remote procedure calls (RPC) to attain the separation between these.

2.2.2 Characteristics

According to Nadareishvili et al. [Nad+16, p. 7] a microservice can be identified by these important characteristics:

Small in size As a result of the segmentation of software by contexts, small services arise. These exclusively cover one purpose and by that only come in small size compared to software monoliths. To reduce complexity for the developers, microservices can generally be owned by separate teams. Thereby microservice measurements can be somewhat determined by team size.

Messaging enabled With the modularization of software into different services, an opportunity of deploying these components on different systems emerged. To consequently asure correct behaviour of the application, a microservice has to provide an interface for communication or respectively for messaging in order to exchange information with the other components of the microservice architetcture.

Bounded by contexts As the microservice architecture to some extent follows the *Domain Driven Design* pattern, the concession of software components with similar properties to a "bounded context" results in a microservice, comprising its function. As a consequence every microservice is defined by one context only and does not interfere with other microservices and therefore not with other contexts.

Autonomously developed As a key feature this pattern enables teams to implement each microservice separately without having impact on the functionality of the other services. With well-defined contracts to other microservices the responsible team takes care of the correct behaviour of the service, as well as provisioning of an interface for communication. This way of implementation contributes to a more beneficial solution, regarding agile software development and Development Operations (DevOPS).

Built and released with automated processes Continuous Integration CI is a development practise where members of a team integrate their work frequently, whereby each integration is verified by an automated build to detect integration errors by including tests [FF06].
With the segmentation of the software monolith, a need for validation of each unit before deployment derives and results in the consideration of a integration technique like CI, that assures the correct behaviour of the independently developed units.

Independently deployable Since every microservice is segregated completely from the other microservices within an application, the transition of getting new features into production gets faster and provides more flexible options for piloting and prototyping Nadareishvili et al. [Nad+16, p. 16]. With DevOPS deployment techniques, like Continuous Integration (CI), only tested units get deployed into production and therefore do not interfere with the application's stability.

Decentralized Since every microservice is a independently deployed unit and the communication between these is typically done via network calls, they might be distributed among different servers or virtual machines.
With respect to software development, decentralization means that the bulk of work done within a system will no longer be managed and controlled by a central body. Consequently the implementation of software changes become easier, faster and will result in fewer bottlenecks and less resistance to change [Nad+16, p. 8].

2.2.3 Structure

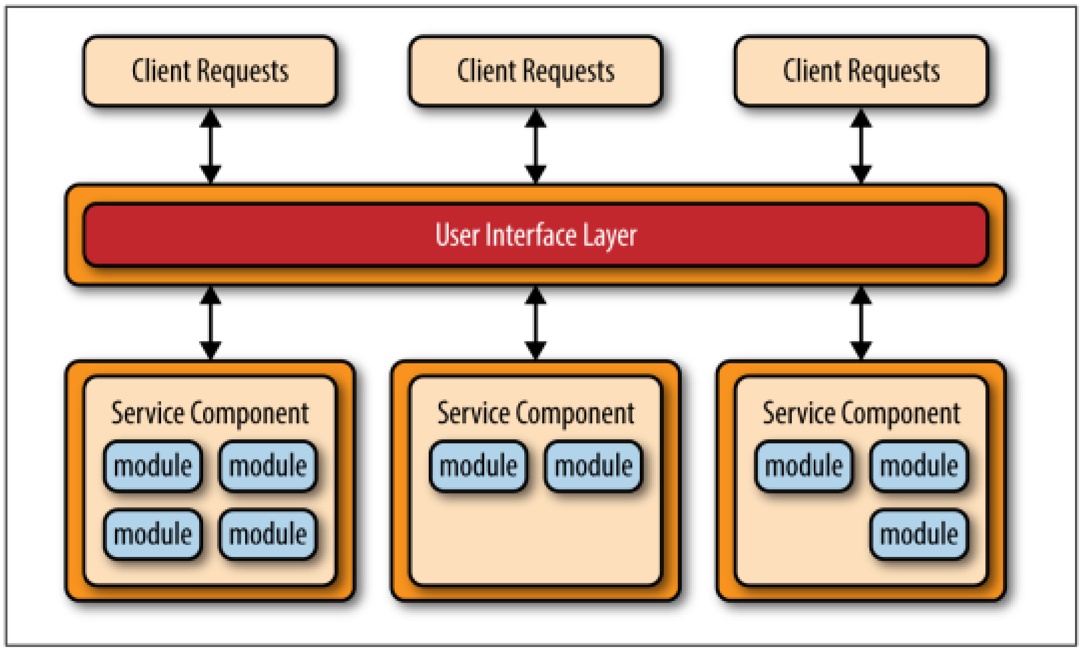
The composition of software modules to microservices is generally determined by applying the *Domain Driven Design* and therefore determines which mircoservice within the system should implement which domain [Wol16, p. 100]. The key concept within the microservices pattern is the distributed architecture. microservices are completely separated from each other and therefore have to communicate via network calls. As shown in figure 2.4, related modules are combined to service components which represent microservices.
Every microservice serves one purpose and gets in general accessed by an *User Interface Layer* via remote access protocol (e.g., AMQP, REST, SOAP, etc.) [Ric15]. Client requests can be user interactions or other systems interacting with the microservice system and are typically done via *Application Programming Interface*. The system itself takes care of the correct orchestration or choreography by either communicating with each other over these interfaces or publishing and consuming Events in order to maintain the core functionality.

Two architectural characteristics that all microservice implementations have in common are *Loose Coupling* and *High Cohesion*:

Looslely coupled As mentioned before, they are loosely coupled and may be distributed across various systems. Each microservice should have few dependencies on other microservies to facilitate the modification of them, since change will have only have an impact on other individual microservices [Wol16, p. 102]. Breaking up systems and identifying bounded contexts in a system by applying Domain Driven Design is an effective way of designing microservice boundaries [Nad+16, p. 64]. Therefore the boundaries of microservices align to some extent to the *Bounded Contexts* of a *Domain Driven Design*.

High cohesion Another aspect is the high cohesion within a microservice. With the definition of *Modules* and *Services* within a process boundary an option to keep related code together emerged. To reduce complexity of a software application, modules with related behaviour should have a close relationship within a *Bounded Context* (see 2.1.2). This ensures that change can be done at one place and avoids deploying lots of services at once when the same functionality is distributed along several services [New15].

Figure 2.3: Basic microservices architecture pattern [Ric15]



2.2.4 Communication

The Communication between microservices can either be synchronous or asynchronous. Synchronous messaging lets the components of this architecture communicate in a blocking manner, resulting in simplicity regarding the implementation, maintenance and collaboration of the system.
On the other hand it might affect the latency or respectively the "responsiveness" of a system, whereas asynchronous communication seems to solve this problem. Asynchronous messaging enables the system to react quickly to *Events* among the components but is difficult to implement, monitor and test.

Each collaboration pattern is based on another model. While synchronous messaging follows the *Request/Response* model, asynchronous communication is more concerned about *event-based* collaboration [New15, p. 89].

Request/Response Model Implementations of the *Request/Response* pattern, like Representational State Transfer (REST) and Remote Proceedure Calls (RPC), have in common that a client initiates a request to an Application Programming Interface and has to wait until the remote process, that handles the request, is finished. The end of the remote process is signaled by a response to the client.

Event-Based Model In Event-Based Models microservices do not know the existence of other components of the system. They rather wait for Events to happen, react to them and may moreover publish further Events. High decoupling and scalability gets achieved by the opportunity of adding new subscribers to events without the event publisher ever needing to know [New15, p. 89].

Another aspect in terms of architecture considerations is the decision to implement *Orchestration* or *Choreography* [New15, p. 90]. Both describe a style of collaboration between components of a system.

Orchestration Orchestration relies on a central component, the Orchestrator, that controls the behaviour of the microservices. This component acts as a supervisor and coordinates the whole process flow of the system by interacting with each component. With this type of service composition every service within the system only cooperates with the Orchestrator in order to achieve a specific domain functionality. As a consequence microservices only have access to their domain and therefore have a limited scope.

Choreography Choreography describes the sequence and conditions in which data is exchanged between participants in order to meet some useful purpose [BK04]. The core focus of this approach is on implementing domain logic by decentralizing the communication to a set of peers or respectively microservices. In comparison to the *Orchestration* the services do not need a central component in order to implement desired behaviour. They rather organize themselves by interacting with each other in an synchronous or asynchronous manner. With the implementation of Service Choreography the system are significantly more loosely coupled, as well as more flexible and amenable to change [New15, p. 92].

2.2.5 Analysis

The *microservices Pattern* may come with some key benefits compared to the software monolith [New15]:

Technology Heterogeneity With the isolation of each microservice the possibility to use different technologies for each service emerged. Teams that collaborate on microservice architectures, may consider the adoption of technologies that might fit best for their scope. With the segmentation of the system into services, the developers are not tied to old the technology stack, but are free to use other technologies at will [Wol16, p. 56]. They might use different programming languages, database technologies or frameworks in order to achieve an enhancement in terms of performance and changeability [New15, p. 20].

Resilience The segmentation of the system into single isolated components may result in more robustness of the application [Wol16, p. 61]. The service boundaries become obvious bulkheads and enable the isolation of failures along the whole system [New15, p. 20]. In addition, problems in the development process get mitigated by the implementation of *Continuous Delivery Pipelines*. With bringing the code of a microservice only into production, if it passes a chain of test stages, such as *Acceptance Tests*, *Capacity Tests* or *Explorative Tests*, only full functional components, that exclude unexpected behaviour, will be integrated into the productive system.

Scaling Variable amounts of user clients, interacting with the system results in a demand for consistent performance and functionality. With the small size and delimited boundaries of the services, replication and therefore load balancing can be applied to specific microservices to serve this consistency. With the distribution of the services across several servers not only load can be balanced across those machines, it can also reduce latencies by choosing corresponding servers based on geographical proximities to the requesting client [Wol16, p.61].

Ease of Deployment Changes can be easily deployed to the productive system in comparison with the monolithic approach. While in the monolithic system the change of a single line of code requires the whole application to be deployed in order to release the change, the microservice approach requires only the coherent microservices to be deployed [New15, p. 24]. Techniques, such as *Continuous Delivery*, assure the isolation of problems due to the migration of new code to production. The independence of deployed microservies allows the code to get deployed faster and makes fast rollbacks easy to achieve [New15, p. 24].

Composeability With the implementation of microservices, the functionality of the system can be consumed in different ways for different purposes [New15, p. 26]. For instance with the provision of mobile and native web applications, a system composed of microservices can obviously satisfy different needs. With the *Composeability* comes the simplicity of modifying or extending the microservices by simply editing, testing and releasing the code at one place without having impact on the other services due to the fact that the only interaction with other services is done via static web service interfaces, such as REST API.

Replaceability The costs of the migration processes of replacing legacy systems with new systems is facilitated by determining sharp boundaries between the microservices. The replacement of one micreservice with the same behaviour and interface for communication does not have an impact on the overall functionality of the system. With the small size of the individual services the costs of deleting them or replacing them with better implementations will be reduced [New15, p. 27].

2.3 JValue ODS

2.3.1 Definition

JValue Open Data Service is an open source software, that serves as a platform for *open data* consuming applications. It provides an web user interface for creating and maintaining scheduled requests for open data from various web interfaces. By giving the possibility to select the data representation (regardless of the format of the data from source) and the possibility to transform the imported data by defining customized "transformation/filter" rules, the open data service supports open data consumers, who encounter problems with the format of the original data source, such as the unsuitability of the amount of data or the format.
The platform also acts as an intermediate persistence layer between open data origin and the consuming application. This is achieved, by persisting imported data sets in a database and by providing an interface (REST) for data extraction. Further functionality is given by the capability to inform the user by sending notifications to specific platforms [1] after successful pipeline execution. The user can define customized conditions, that the data has to meet, in order to trigger the dispatch of the notification to the corresponding platform.

2.3.2 Structure

The architecture of the service follows the *microservice Pattern* with a *Request/ Response* messaging model. It is composed of several *Docker Containers*[2] that communicate via HTTP-Requests on Representational State Transfer (REST) APIs. Each service is placed in a container and serves another purpose, such as the provision of an web user interface or the data persistence.
The communication pattern is implemented as orchestration, whereas the scheduler acts as a central component that controls the process flow within the system. This is done by sending raw data or structured configurations in SJON format via synchronous HTTP-POST requests and the receiving corresponding results from each service for further processing (e.g. passing it synchronously to another microservice). Development is done (in collaboration with all contributors) via *Continuous Delivery*. This means that every release has gone through at least one test stage in order to check the integrity of the service.

[1]currently supported platforms: slack, firebase and custom webhooks
[2]https://www.docker.com

2.3.3 Components

Web-Client This component provides an web user interface (UI) for easy and seamless configuration for the import from *Open Data Sources* and for *Pipelines*. The user can manually configure the system's plan when to fetch data from the original data sources to counteract quota restrictions or scheduled downtimes. It furthermore provides the functionality to test and evaluate transformation code for data aggregation (in javascript syntax) with an embedded "notepad-like" interface. The entered transformation code will be used by transformation service for further aggregation of the imported open data.
The technology used for this service is Node.js [3] with various frameworks, such as Vue.js[4] and typescript[5].

Core-Service The Core-Service is responsible for the persistence of *pipeline configurations*. It furthermore exposes an interface for the organization of these configurations via REST API, that can either be accessed directly by the user/ application or indirectly by using the user interface of the *Web-Client* service.

The pipeline configuration consists of several parts that can manually edited by the user directly through interaction with the user interface or via REST API:

> **Adapter Config** Within this configuration, the user can provide the location of the open datasource and the data representation (e.g. SJON) as well as additional information, such as license or data descriptions to the system.
>
> **Scheduler Config** The *Scheduler Config* contains information of the times the open data has to be requested from its source. It can be configured as a "one time only retrieval" or as a recurring event (e.g. hourly).
>
> **Transformation Config** The *Transformation Configuration* is a code script (based on javascript) that is evaluated by the transformation service. The user has therefore the possibility to enrich, filter or aggregate the imported open datasets.

[3]https://nodejs.org/en/about/
[4]https://vuejs.org/
[5]https://www.typescriptlang.org/

Notification Config With the *Notification Config* the user can indicate whether to be informed when open data is received and available within the system of the JValue ODS. Users can choose between various platforms, such as Slack[6] and FireBase[7] to be notified on or can configure individual webhooks. In addition a condition (javascript based) can be provided that gets evaluated after successful transformation. If the condition is met the notification will be delivered to the corresponding platform.

The *Core-Service* utilizes a postgres database for pipeline configuration persistence and is implemented in Java with springboot framework[8].

Scheduler-Service The Scheduler-Service is a component that orchestrates the executions of pipelines. It is a central component, that is responsible for the correct process flow within the system, which is composed of distributed services. On frequent basis, it extracts pipeline configurations from the *Core Service* and caches them locally. With its integrated scheduling functionality, it is capable to trigger a sequence of actions according to the time that is defined in the scheduler configuration of a pipeline.

After the retrieval of the open data by the *Adapter microservice* it redirects the data to the *Transformation Service* that filters and transforms the received data to the desired format. The resulting data gets persisted by the Storage-Service and a notification will be sent by the *Notification Service* (if configured). The Scheduler Service acts as a central component which triggers these consecutive actions and takes care of the intercommunication between the microservices.

The technology used for this service is Node.js with typescript framework.

Adapter-Service The purpose of this microservice is to retrieve raw open data from external data sources over *Hyper Text Transfer Protocol* (HTTP). With the provision of the *Adapter Configuration* to its REST interface the *Adapter-Service* retrieves information of the location of the open data source as well of the desired resulting data representation (e.g. JSON). It provides an REST interface which returns the requested data by providing an *Adapter Config* to its REST application interface via HTTP POST request. The *Adapter Service* is implemented in Java with springboot framework.

[6]https://slack.com
[7]https://firebase.google.com/
[8]https://spring.io

Transformation-Service The purpose of the transformation service is to aggregate the imported open data according to the corresponding transformation config of the same pipeline. By presenting data and transformation configurations to its interface via HTTP-Post requests, it will evaluate and transform the data. The resulting aggregations of the data will be returned as a HTTP response.
It furthermore provides an interface for dispatching notifications to various platforms, such as slack. With the supply of transformed data and a notification config, it will evaluate the condition that is embedded in notification config with the provided data. If the condition is met for the data, it will trigger a notification to the corresponding platform.

The underlying technology is Node.js with typescript framework.

Auth-Service The authentication service adds a layer of security to the system. The user has to authenticate to it in order to get access to the resources in his scope (e.g. to created pipeline configurations). With this service the system is capable to achieve access control in an efficient way. Every service is connected to this service for maintaining controlled utilization to their interfaces.

Storage-Service The *Storage-Service* is the last stage of the life cycle of open datasets. After import by the *Adapter-Service* and further aggregation by the *Transformation-Service* the transformed data gets persisted in its database of the *Storage-Service*. The bounded context of this service is composed of two aggregates. One is the database itself and the other is a service, providing a REST interface to operate on the data in the database. The *Auth-Service* thereby ensures that the user can only access with read privileges.

The chosen technology for the database is postgres[9]. For accessing the data via HTTP-Requests a postgrest [10] docker container will be deployed.

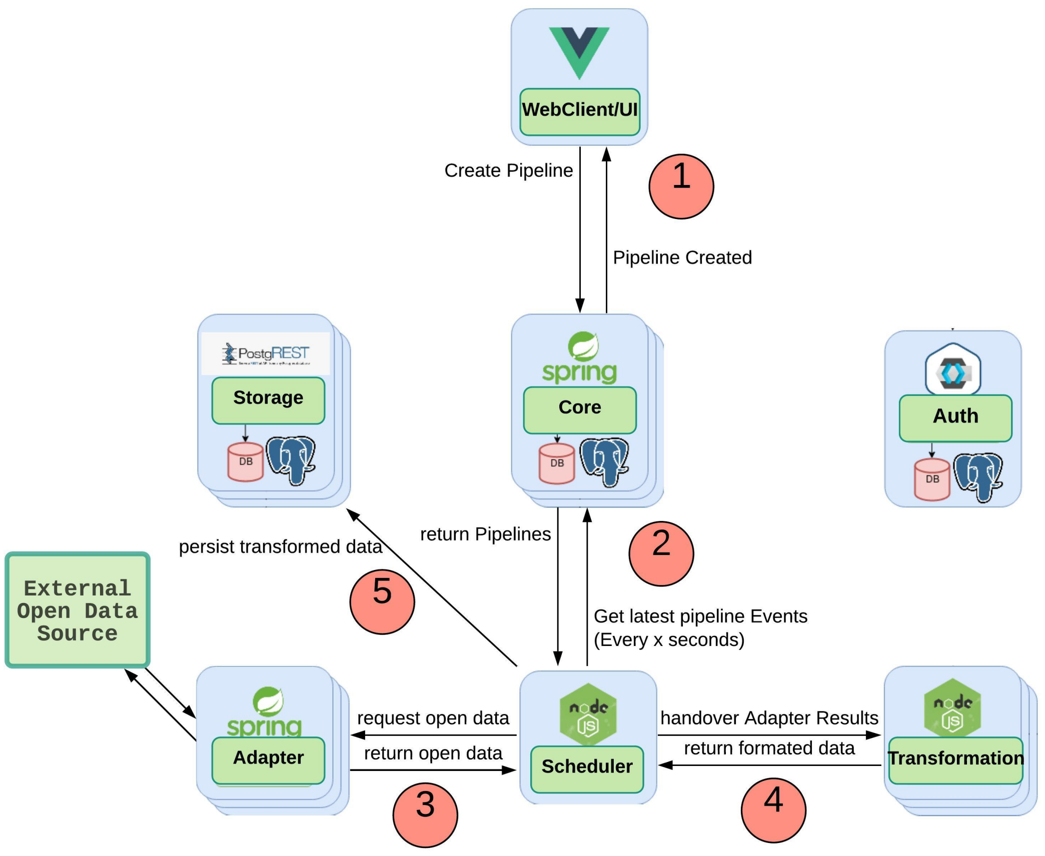
Reverse-Proxy Due to its decomposed structure, it might be possible that the locations of the services are distributed or may change. Therefore the reverse proxy is a entrypoint for the user to interact with the system's components. It can also be used for inter-service communication.

[9]https://www.postgresql.org/
[10]http://postgrest.org

2.3.4 Process Flow

This section describes the lifecycle of a pipeline. It shows all the actions to be taken by the system in order to deliver open data in a desired form to the user or respectively application. The communication pattern is an orchestration with the Scheduler-Service as orchestrator. The communication between the services is completely handled by the *Core-Service* by pulling data from one service and pushing to another. It relies on a synchronous request/reply model on REST interfaces, whereas the core component is the only communication partner for each microservice.

Figure 2.4: Lifecycle of a pipeline



With the creation of a pipeline (Step 1) on the *Web-Client* a request, containing the pipeline configuration in JSON format, is sent to the *Core-Service*.
This service persists it to the database and replies to the Web-Client, indicating the result of the operation. The *Scheduler-Service* requests frequently pipeline configuration deltas from the *Core-Service* and holds or respectively updates them

in its local cache (Step 2). It loads the scheduling configuration to its scheduling module and triggers the execution of the pipeline when the time criteria is met. Upon pipeline execution the *Scheduler-Service* sends the local cached adapter configuration to the *Adapter Service*, which will then import open data from the external source and will respond with the extracted data as payload (Step 3).
The *Scheduler-Service* will publish the extracted data and the corresponding transformation config to the *Transformation-Service*. This service takes care of the transformation and will result in replying with the transformed data (Step 4).
The scheduler will send the transformed data to the REST interface of the *Storage-Service*, where the data gets persisted (Step 6).
As last step it will send the transformed data and the notification config to the *Transformation-Service* which will evaluate the condition in the config with the transformed data as input and send a notification to an external platform on evaluation success.

2.4 Event-Driven Architecture

2.4.1 Definition

Event-Driven Architecture (EDA) describes a pattern in which interaction or respectively messaging between components of a distributed system is done by using *Event Notifications*. It can furthermore be understood as an asynchronous messaging approach for systems consisting of highly decoupled components, like microservices. Systems, that implement this pattern, include units that produce, transmit, process and consume events.

An Event can thereby be interpreted as an occurrence or happening, which originates inside or outside a system and can be consumed by a system's component [HPX13, p. 13]. An Event Notification on the other hand is a message that informs interested recipients that an Event happened. It can be moreover defined as an event-triggered signal sent to a at run-time defined recipient [Fai11]. In the context of EDA Event and Event Notification might be considered as interchangeable terms.

Event-Based Systems consist of *Event Producers/Publishers*, that emit *Events*, such as system state changes and *Event Consumers*, which represent interested parties that evaluate the Event and optionally take action based upon the type of the Event. An important note to take is that event publishers do not know which part of the system consumes the emitted events, as well that event consumers have no information of the origin of the events, they subscribed to.

This pattern has the potential of easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale [MFP06].

2.4.2 Elements

Events The structure of events can be subdivided into event header and event body. The header contains meta information that describes the event, such as the event type, event name, event timestamp and event creator, whereas the body contains relevant payload that needs to be published and consumed.
Events can be primitive and composite. While the former describes occurrences within the system, which are atomic and occur only at one point in time, the latter is composed of several primitive events that occur over time and may have specific patterns [HPX13, p. 13].

Event Producer An event producer is a component that publishes events to an event driven system and is often denoted as “publisher“. One key concept is, that it does not address a specific receiver. Its implementation can be moreover interpreted as “self-focused“ and does only observe its own state. It will rather forward it to a notification service, for instance rabbitmq[11], which will take care of the correct distribution of the published events [MFP06, p. 12].

Event Consumer Event consumers interact with elements of a notification service, such as *event channels*. They indicate interests in specific system state changes by issuing *subscriptions*[MFP06, p. 12]. A subscription can be described as a set of events to which consumers can show interest by subscribing.
Events are generally published with the intention to trigger further processing actions. This can be achieved either by the consumer itself or by delegation by the consumer, whereas the consumer also can act as a producer.

Channels An event channel portrays a mechanism to propagate system state changes from the producer to interested parties and can be furthermore understood as a subscription model [Fai11, p. 91].

A channel is composed of these elements [EN10, p. 135]:

> **Event channel identifier** Every channel can by identified by a unique name, that is provided to it.
>
> **Terminals** Each channel has an input and an output terminal. The further is used to publish events to the queue while the latter is for event consumption.
>
> **Routing scheme** A channel makes routing decisions in order to route events that have been demanded by subscribing to event subscribers.
>
> **Quality of service assertions** These assertions specify non-functional characteristics, such as security, performance or availability aspects.

[11]https://www.rabbitmq.com

Routing Routing can also be interpreted as a filter mechanism for channels and is therefore often denoted as filtering. It represents a set of rules for a channel to make a decision where to send incoming events, whereby the events remain unchanged.

Events can be routed by using one of these mechanisms:

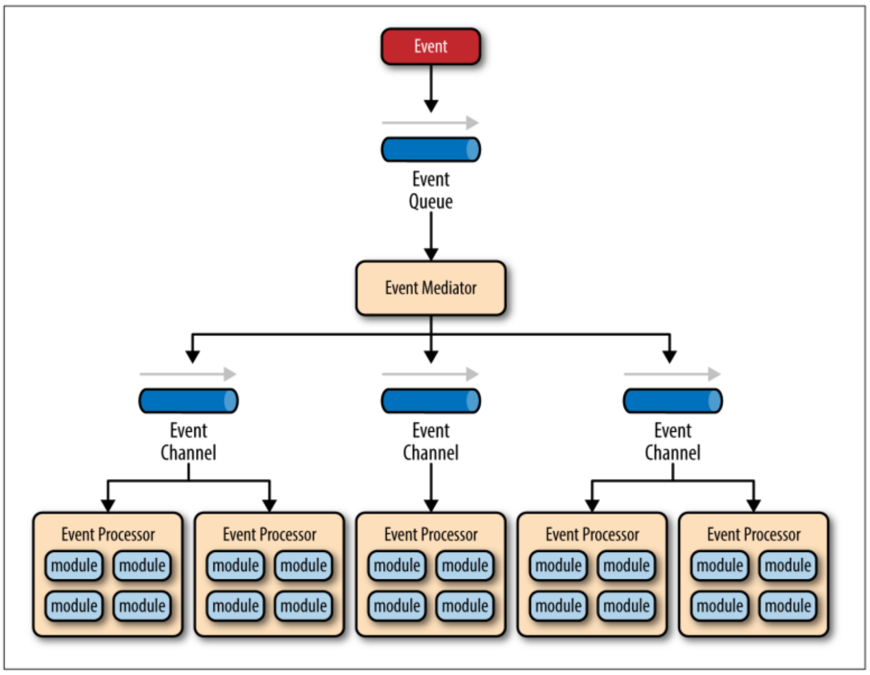
> **Fixed** This is the simplest for of routing, where no filtering is applied. The channel routes every event that is published to it to the output terminal, with the consequence that every consumer that subscribed to that channel will receive all published events [EN10, p.136].
>
> **Subject-Based** This routing type uses string matching, whereas publishers annotate each event with a subject string, that denotes a rooted path in a tree of subjects [Fai11, p. 19]. "/University/fau/tecfac/oss" can be such a subject path to identify location where an event can be published to. Consumers can then subscribe for instance to the path "University/fau/*" in order to get all events that have been published in that tree structure.
>
> **Type-based** With this form of filtering, the channel utilizes the event types to make routing decisions. Path expressions, as well subtype inclusions will be therefore used, resulting in routing of to the same node of a routed path (such as in subject based filtering) [MFP06, p.19].
>
> **Content-based** With content-based routing, the whole content of an event will be evaluated. If a the event contents match with a specific conditional expression, it will be routed to the corresponding location. This can be achieved in various ways, for instance by template matching, simple comparisons or XPath expressions in XML [MFP06, p. 20].

2.4.3 Topologies

Mediator With the mediator topology the system is composed of multiple event consuming and producing software components that rely on the orchestration by a central unit, the mediator. The mediator takes care of the order of events that have to be processed, as well as of the communication between event producers and event-processing consumers.

The type of events in this pattern can be categorized in an initial event and a processing event. While the an initial event denotes the occurrence of a system state change that might lead to actions within the system, the processing event is sent to processing components to imply that an action has to be taken [Ric15, p. 12]. The role of the mediator is to consume the initial event from a channel and publish processing events to the right event processors.

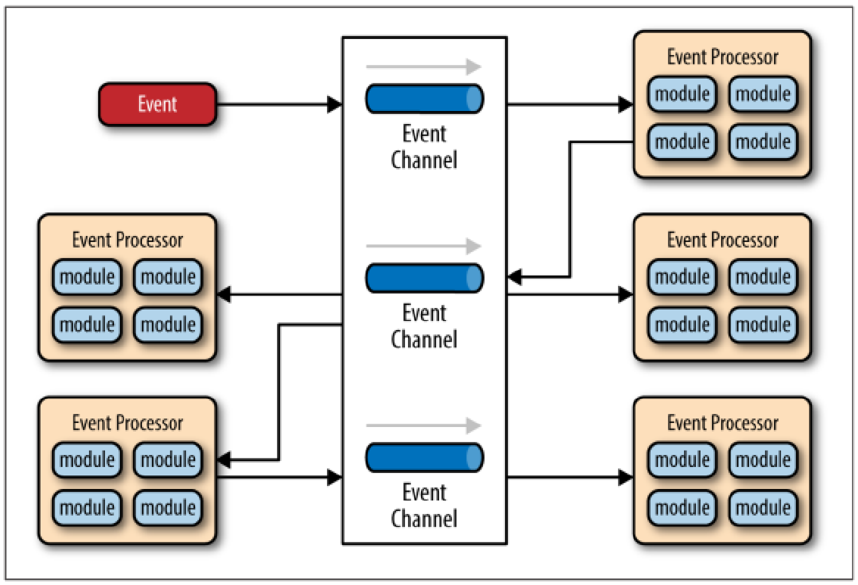
Figure 2.5: Mediator topology example [Ric15].



Broker In contrast to the mediator, the broker topology follows an orchestrated communication pattern. This means that the message flow is rather distributed across the event processors in a chain-like fashion, than controlled by a central component [Ric15, p. 14].

The event processors thereby take care of the consumption of the event that illustrates a system change and performs an action. They furthermore indicate that an action has been performed by publishing a new event to the system. The broker therefore only serves the purpose of providing the technology to enable the communication between event publisher and event processor (which can be the same component).

Figure 2.6: Broker topology example [Ric15, p. 14].



2.4.4 Event-Based Patterns

Competing Consumer The *Competing Consumer* pattern describes a mechanism to efficiently distribute events that trigger processes among subscribers of a Point-to-Point Channel or respectively a “fixed“ routing scheme (see 2.4.2) The parties that issue a subscription to the same event, will be served in a “fist come, first serve“ manner. Due to the fact that an event is being consumed upon receival, it is no longer available for the other subscribers. This procedure reduces bottlenecks and increases efficiency by letting only those consumers take event processing actions who have available resources for the consumption and therefore for the processing [HW12, p. 446].

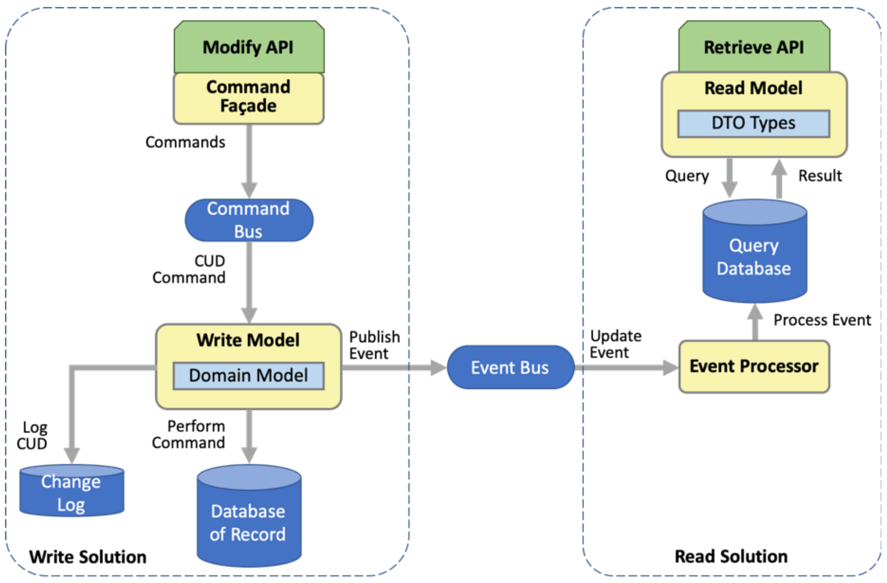
Event Sourcing Event-sourcing captures the state of a domain object as a sequence of state-changing events [Ric20a]. By changing the state of an entity, a new event is appended to a list of previous occurred events rather than just replacing the state of this object. This sequence of event is persisted to an event store, which can be interpreted as an database for events. It exposes persisted events to all interested parties and can furthermore be used to reconstruct the state of an domain object by just replaying the events from a specific state (initial or snapshot) [Ric20a].

Saga The *Saga* pattern is pattern for transactions that span across distributed components of a system with the intention to guarantees consistency. A saga can be understood as a sequence of local transactions, whereby a successful execution

of such a transaction triggers another transaction located another component. If a local transaction fails due to the violation of business constraints, compensating transactions that undo the changes are chained back to the component that initiated the saga in order to maintain the original state and consequently consistency across the distributed services [Ric20b].
In an event driven context, the saga is implemented as an event based choreography. Each local transaction publishes events to the succeeding component that causes another locatl transaction.

CQRS Command-Query Responsibility Segregation (CQRS) is a pattern that strictly segregates operations that read from operations that write data to a database with the intention to scale and perform better, as well as to enhance security [al20a].
The architecture consists of a "Query Database" that only serves the purpose to provide its datasets via "Retrieve API" and "Database of Record" that is designed for creation, and modification and deletion of data via "Modification API".

Figure 2.7: Architecture - Command Query Responsibility Segregation [al20a]



The "Write solution" as shown in figure 2.7 handles "Create-Update-Delete"(CUD) operations of domain entities (see 2.1.5) that get applied to the database and subsequently published to the "Read Solution" in order to synchronize the data of the "Database of Record" with the "Query Database". The "Database of Record" thereby acts as "single source of truth" [al20a].
The CQRS pattern complements the *Event Sourcing* pattern. An event store can be modelled as change log that keeps track of states of the database and is responsible to publish the change events to an Event Bus.

3 Requirements

3.1 Functional

F1 Choreography instead of orchestration The system should move from the existing orchestrated communication pattern to a event-based choreography. As process flow of the system is currently determined by the control of a central component, the scheduler, a need for the replacement of the orchestrated communication pattern with event-based choreography emerges.

F2 Asynchronous messaging All synchronous request/reply patterns should be replaced with asynchronous event messaging. Due to the blocking behaviour of the request/reply model that is implemented in the open data service, the switch of all these communication mechanisms to asynchronous event-based message exchange is required

F3 Inter-service dependencies A microservice should be only depending on the message bus when running and not on other microservices.

3.2 Non-Functional

N1 Functionality of the system has to be preserved The behaviour of the system should not change to the outside, so that ODS users will not recognize that a change has been made to the system. The Invariant of the system should stay unchanged.

N2 Increase in performance The overall system should increase the throughput of the system (amount of parallel running pipelines).

N3 Collaboration and Development The changes to the architecture should be planed in meaningful iterations, that are all applicable to the repository [1], without interfering the functionality. In Addition drastic changes have to be discussed with the contributors of the software,

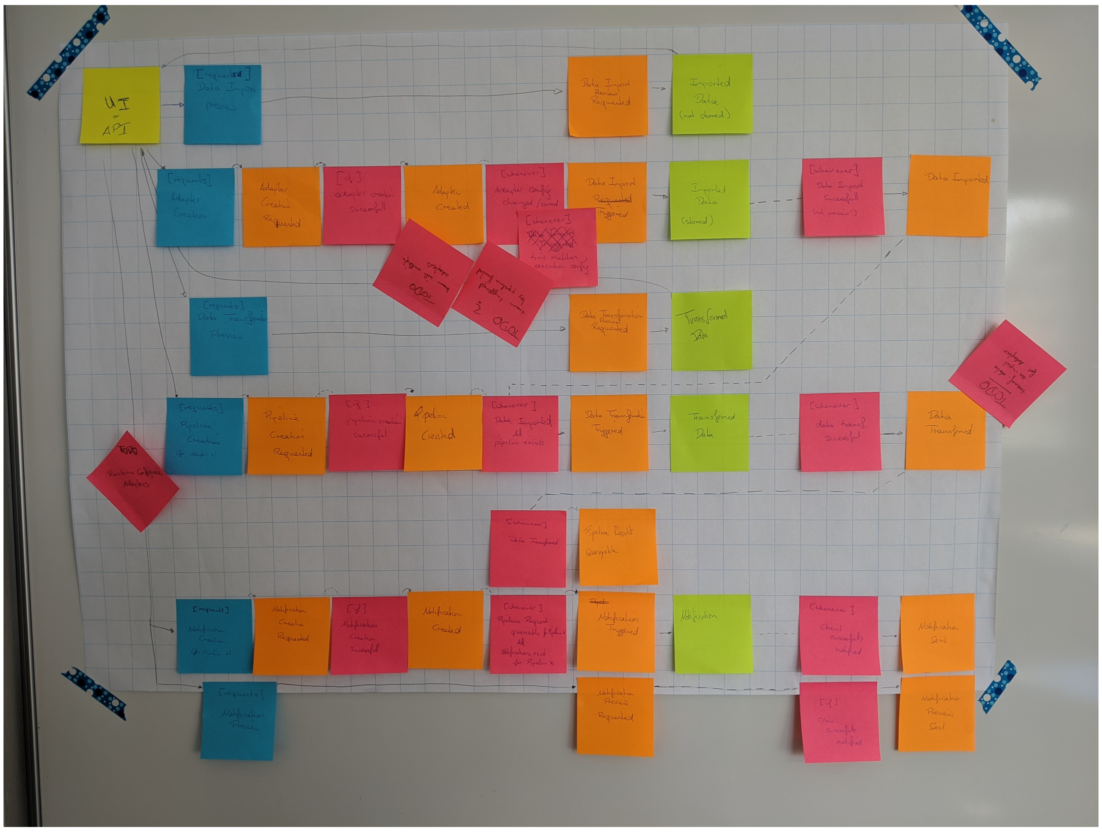
[1]https://github.com/jvalue/open-data-service

4 Architecture Design

4.1 Analysis with Event-Storming

To identify the events and to reevaluate the architectural design of the current system, a technique, called *Even-Storming*, was applied. Therefore an *Event-Storming* workshop was held in which a part of the *JValue ODS* developers or respectively constributors participated.

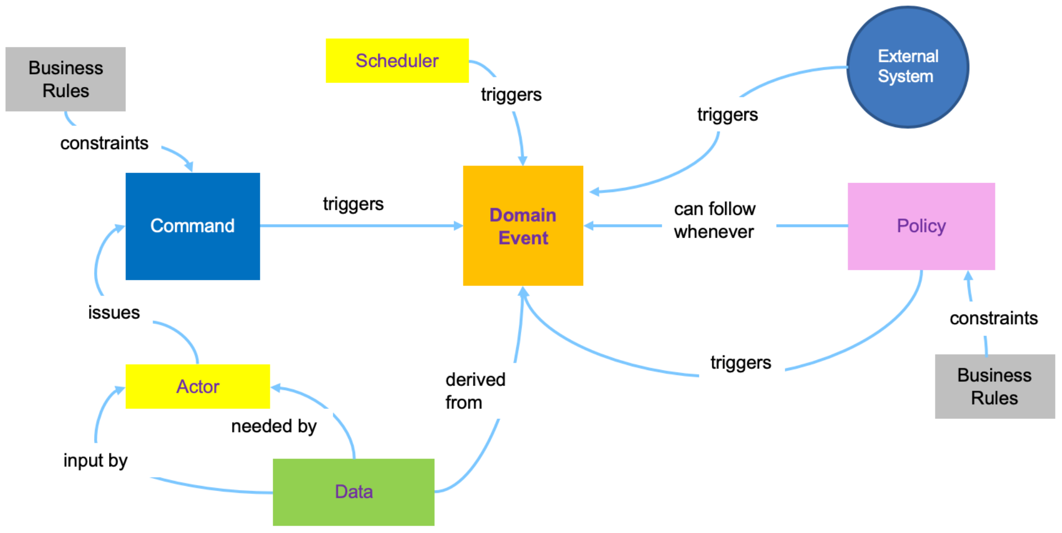
Figure 4.1: Event-Storming Workshop: results



4.1.1 Definition

The *Event-Storming* technique was invented by Alberto Brandolini with the intention to facilitate the creation of system architectures based on *Domain Driven Design*. In order to get a better understanding of the business problem, stakeholders (including developers, product owners and domain experts) of a specific domain are invited the workshop. This approach can also apply to identify bottlenecks in a system.
"Sticky notes" are used in order to illustrate different types of building blocks in DDD. Besides reevaluating the current structure of the system in a domain driven context, the main draftback of this workshop should be that all events between the different entities of a Domain (2.1.1) will be identified and brought into context.

Figure 4.2: Event Storming - Building block relations [al20b]



The entities that are used in this workshops are "sticky notes" which represent the following building blocks [Bra19].

Domain Events Domain Events indicate that a system state change occurred. Their existence within the system may have different origins. They can be a result of a user interaction, triggered by external systems, triggered by time or as a result of some cascading reaction.

Data Data will be presented to actors or derived by domain events.

Actor An actor can be a user or an application that interacts with the system. They may use data to make decisions and issue commands. Actors can also trigger Events directly.

Command Commands are the entities which will result in a system state transition and have therefore a domain event as consequence.

Policies Policies represent some logical rules that exist within a domain. They might be considered as rules that may trigger an domain event with interaction by an actor.

External systems Systems outside of the domain may interact with the system and therefore can trigger events.

Pivotal Events and Swimmlanes The horizontal axis on the wall represents the time, whereas the vertical one can be used to define context in regards of events. Pivotal events are domain events that can be used to indicate a major change to the system (from a temporal perspective). Due to the fact that many events in a domain may occur simultaneously, a building block called system candidate boundary can be used to separate events by different aspects of the domain, resulting in different (swimm-) lanes.
The usage of both elements facilitate the identification of *Bounded Contexts* (see 2.1.2) and as a consequence the identification of microservices.

4.1.2 Procedure

The workshop was split into several iterations [al20b].

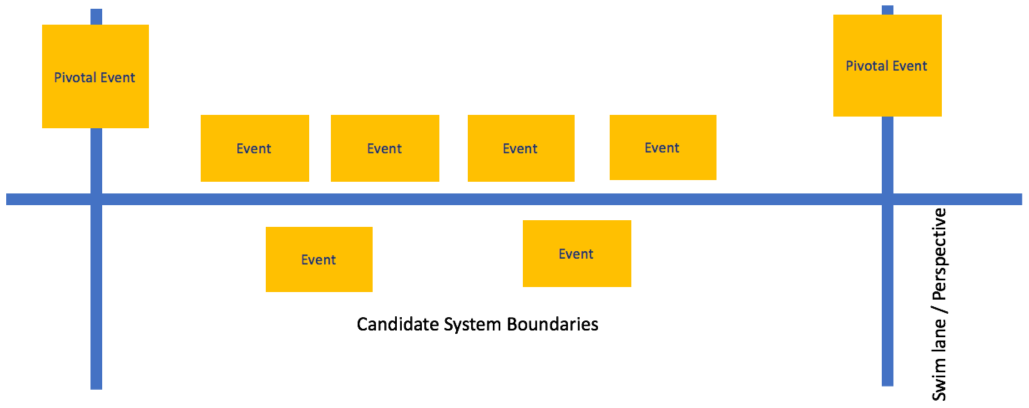
Domain event discovery The first step of the workshop was to identify the current events of the system and bring them in the right order. This was done by putting all "sticky notes", which represent domain events, on the wall. After a period of time all important events were identified and were consequently brought into the right sequence.

Tell the story The next step was the reevaluation of the previous step. The sequence of events was told as a story that occurred within the domain. In the context of the open data service, the story from the configuration of a pipeline to the extraction of the transformed data was told from different perspectives.

Find the Boundaries The goal of this iteration was to find time boundaries indicated by pivotal elements on the one hand (see figure 4.3), on the other hand to identify subject boundaries, that can be detected when multiple simultaneous series of events that only come together at a later time occur [al20b].

These boundaries will be used to identify bounded contexts and consequently microservices.

Figure 4.3: Event-Storming: Boundaries [al20b]



Locate the commands In this iteration the origins of all events were identified. Events may occur when commands are issued within a system or when conditions of policies of the domain are met. In addition they can be triggered by a scheduler or when external systems or sensors provide a stimulus [al20b]

Describe the data In this step the data that is used in the domain was introduced to the model. Therefore “sticky notes“ that represent data with the corresponding description were placed near the previous modelled building blocks.

Identify the Aggregates Aggregates emerged from the process by grouping related events and commands and suggest microservice boundaries and are most likely defined by pivotal evens and candidate system boundaries [al20b]).

4.1.3 Results

The results of the event storming workshop can be found in Appendix A.

Lanes and aggregates The resulting lanes, and microservice boundaries differ to some extent from the previous architecture.
A new lane with the topic ”notification” was introduced. All events (and other building blocks) that are related to the functionality of sending notifications to

another platform were put in the "notification" context. These events were previously a part of the transformation service (see 2.3.3) but represent another aspect of the system and should be therefore defined in an own context.
Another outcome is that there exists no lane that represents the scheduler service (see 2.3.3). It belongs moreover to the adapter aggregate, due to the fact that the event, that a specific time occurred, is only consumed by the adapter service.

In addition storage was not considered to be a an own aggregate and assign to the transformation lane, due to its purpose to persist transformation results.

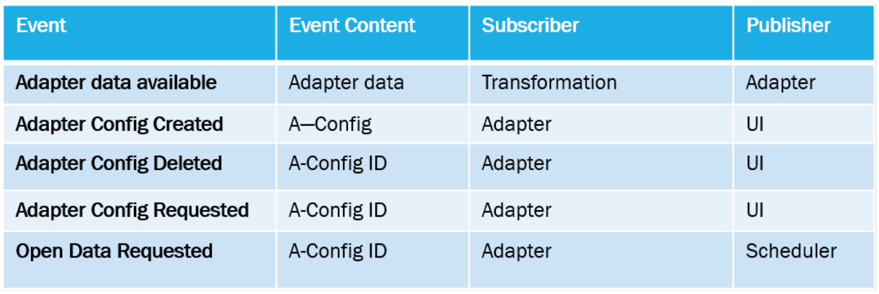
Core-Service Due to the fact that an architecture for the open data service already existed, previous elements were taking into consideration, resulting in the lane "pipeline" for the core service (see 2.3.3), even though it might not correctly defined in a domain driven context. The purpose of this service is the persistence and offering of pipeline configurations. The pipeline configuration of previous architecture is composed of adapter, scheduler, transformation and notification configuration. The event storming modeling approach resulted in the insight that each configuration within the pipeline configuration should be located to the corresponding bounded context and therefore in the corresponding lane. The consequence is that every configuration will have an own repository that is attached to the related microservice., resulting in the unnecessity and as a consequence to the elimination of the core service. For example the transformation configuration will not be stored within a pipeline configuration in the core service. It will be furthermore persisted in a repository in the transformation service.
Another outcome regarding the open data service configuration is that the scheduler configuration is a part of the adapter configuration. The reason behind this is that a scheduler configuration cannot exist without a adapter config, meaning that a data import cannot be triggered if a data source does not exist. This embedded configuration structure will be persisted within the adapter bounded context.

Events

Adapter Several events have been discovered within the adapter aggregate. In order to apply event driven architecture to the existing system of ODS only the events that have been triggered from other aggregates in the ODS domain were taken into account (see figure 4.4).

Three critical events exist within the adapter aggregate that are related to the adapter configuration. Due to the reconsideration of the deletion of the core aggregate adapter configs will be stored within the adapter context. The commands to create, modify or delete those imply the existence of corresponding events. Another crucial event that exists within the adapter boundary is an event that indicates that the imported data is available to the system for further processing (transformation).

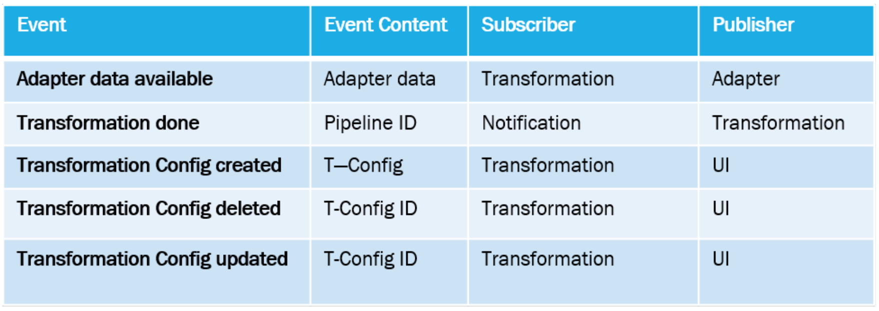
Figure 4.4: Event Storming - Adapter Events



Event	Event Content	Subscriber	Publisher
Adapter data available	Adapter data	Transformation	Adapter
Adapter Config Created	A—Config	Adapter	UI
Adapter Config Deleted	A-Config ID	Adapter	UI
Adapter Config Requested	A-Config ID	Adapter	UI
Open Data Requested	A-Config ID	Adapter	Scheduler

Transformation The transformation aggregate consumes the event “Adapter data available“, that is located to the adapter lane with the consecutive action of data processing (transformation). The configuration related events are similar to the ones in the adapter lane. After the transformation process has finished an event is published to the notification service in order to trigger the dispatch of a notification.

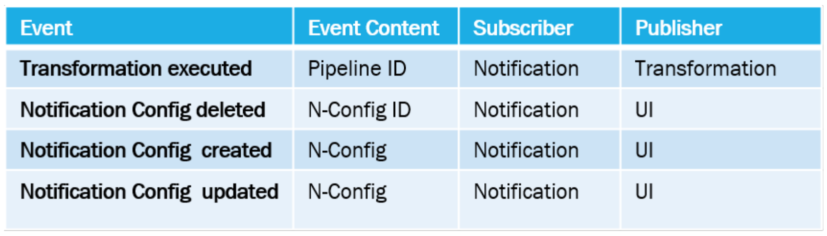
Figure 4.5: Event Storming - Transformation Events



Event	Event Content	Subscriber	Publisher
Adapter data available	Adapter data	Transformation	Adapter
Transformation done	Pipeline ID	Notification	Transformation
Transformation Config created	T—Config	Transformation	UI
Transformation Config deleted	T-Config ID	Transformation	UI
Transformation Config updated	T-Config ID	Transformation	UI

Notification The notification service exclusively consumes events in from the system. It subscribes to events regarding the notification configurations and to the event published by the transformation service to reveal that the transformation process has been finished.

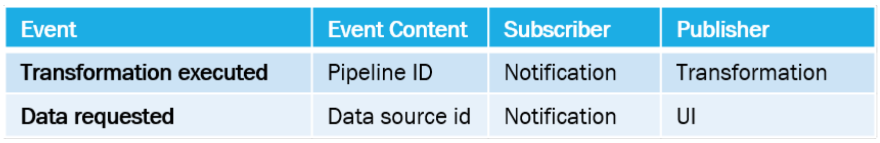
Figure 4.6: Event Storming - Notification Events



Event	Event Content	Subscriber	Publisher
Transformation executed	Pipeline ID	Notification	Transformation
Notification Config deleted	N-Config ID	Notification	UI
Notification Config created	N-Config	Notification	UI
Notification Config updated	N-Config	Notification	UI

Storage Service The purpose of the storage service in the previous architecture was to persist and provide transformed data. As a consequence it is subscribed to the event expresses the transformation data is available, as well to the request of the user interface or user to subsequently respond with the persisted data.

Figure 4.7: Event Storming - Storage Events



Event	Event Content	Subscriber	Publisher
Transformation executed	Pipeline ID	Notification	Transformation
Data requested	Data source id	Notification	UI

4.2 Event-Based Design of ODS

4.2.1 Technology Considerations

Two technologies for event communication where taken into consideration. On the one hand Apache Kafka [1] and on the other RabbitMQ [2]. While the further is an stream-processing software platform developed by the Apache Software Foundation, the latter is based on message queues using *channels*

Apache Kafka The core entities used in Apache Kafka are event streams. They represent a replayable ordered sequence of events, similar *to* the *Event Sourcing* pattern (see 2.4.4), whereas components of a system can publish events and issue subscriptions to it. Kafka can be run on a cluster of multiple servers and is capable of handling large amounts of events. It is often used for real-time processing applications that provide real-time analysis by pattern matching. Due to the characteristic of storing events in a immutable sequence, the commit log, it can also be used as a persistence layer. Besides its consuming and producing Application Programming Interface's, it provides three more API's, such

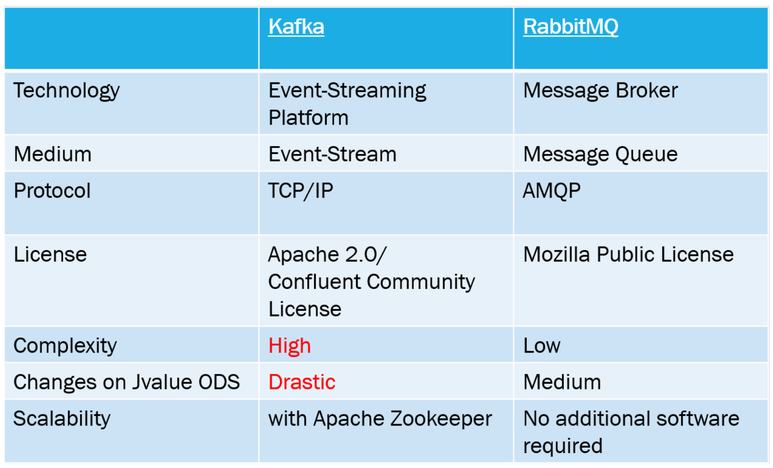
[1]https://kafka.apache.org/
[2]https://www.rabbitmq.com

as a *Connector API* that allows users or respectively applications to connect to additional resources, such as databases or REST interfaces.

RabbitMQ RabbitMQ,whereby MQ stands for message queue, is a message broker system (see 2.4.3) that enables applications to connect and scale. It acts as an intermediate layer of communication for system components to exchange messages asynchronously over so called "channels". As Kafka it can be clustered and therefore scaled. Main subjects, besides channels, are exchanges which are logical entities where events or respectively messages can be sent to, similar to Terminals (see 2.4.2). Another building block in the context of the RabbitMQ message broker are bindings that implement the functionality of routing schemes (see 2.4.2).
RabbitMQ supports a variety of messaging protocols, including the Advanced Message Queuing Protocol (AMQP) [3], that is most commonly used.

Technolgy comparison Both technologies are applicable for the JValue Open Data Service due to the license and capabilities. One major concern with the consideration of applying Kafka to the existing system is the amount of changes that have to be implemented. With this approach every microservice has to be refactored to a great extent, resulting in

Figure 4.8: Kafka vs. RabbitMQ (in the context of the ODS)



	Kafka	RabbitMQ
Technology	Event-Streaming Platform	Message Broker
Medium	Event-Stream	Message Queue
Protocol	TCP/IP	AMQP
License	Apache 2.0/ Confluent Community License	Mozilla Public License
Complexity	High	Low
Changes on Jvalue ODS	Drastic	Medium
Scalability	with Apache Zookeeper	No additional software required

4.2.2 Architecture

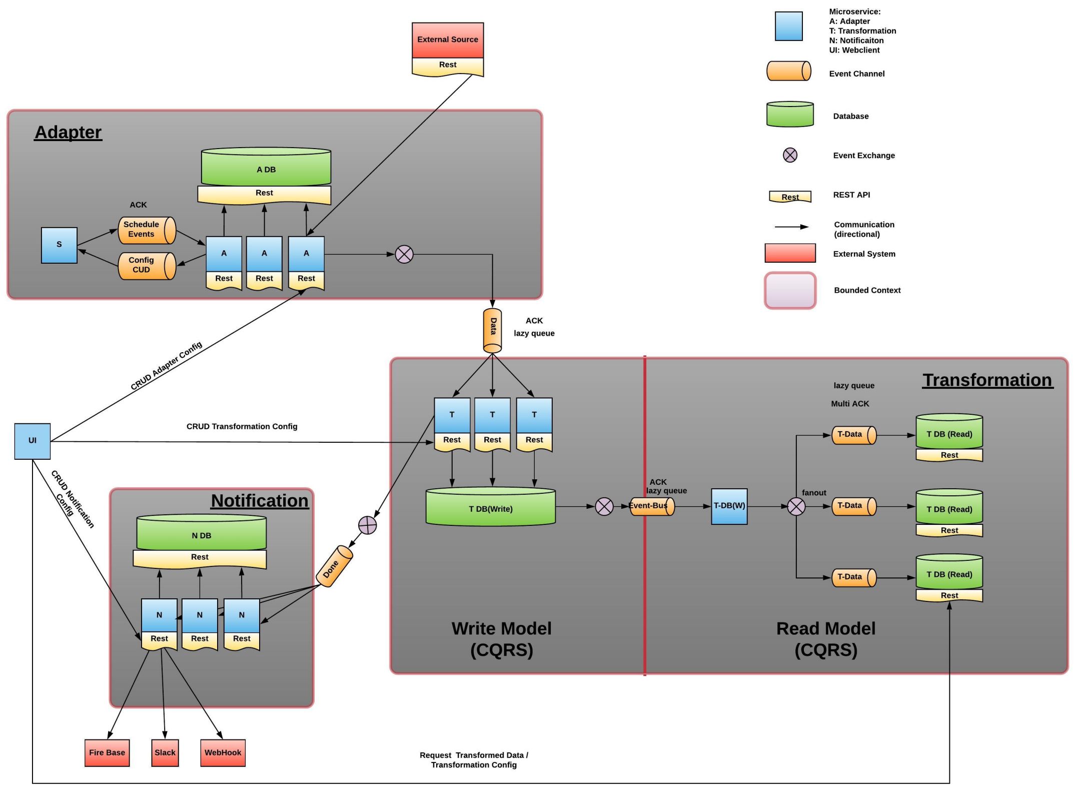
The final event driven architectural concept is derived from the results of the "Event-Storming workshop". The system is composed of 4 bounded contexts as shown in figure 4.9.

[3]https://ieeexplore.ieee.org/abstract/document/4012603

- UI Service
- Transformation Service
- Adapter Service
- Notification Service

The architecture has been established with the intention to scale up the system by increasing the amount of microservice instance with container orchestration software (e.g. with kubernetes [4]). Therefore the services have been modeled with multiple instances, as far as it made sense.

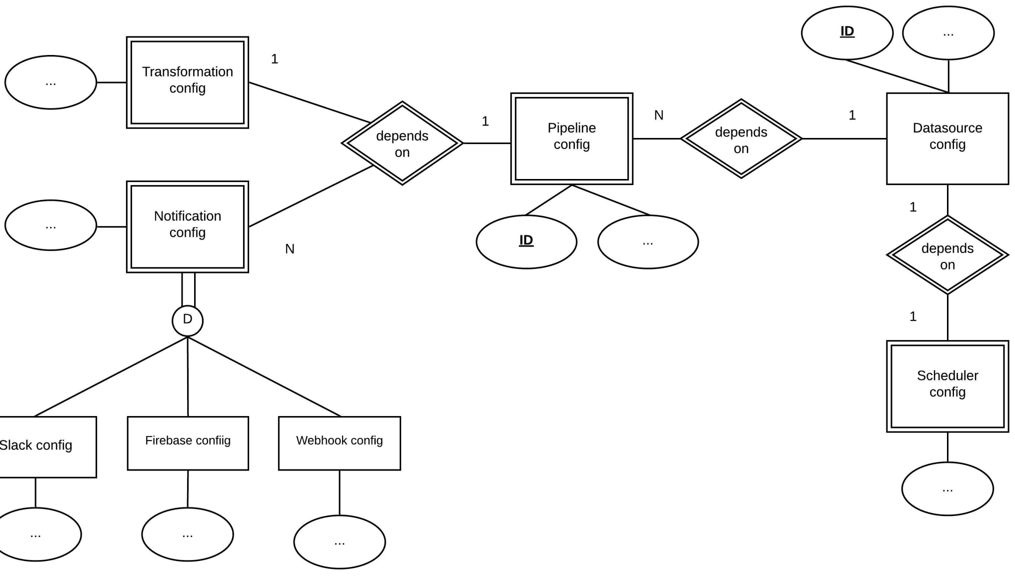
Figure 4.9: Event Driven Architecture of ODS



[4]https://kubernetes.io/

Decomposition of pipeline configuration A pipeline in the context of JValue ODS describes the process from the extraction over the transformation to the persistence of open data with subsequent notification on external platforms.
In the previous architecture the properties of data import (adapter config), data extraction time (scheduler config), data aggregation (transformation config) and notification (notification config) were each embedded within a 1:1 relation in the pipeline config. This results in the logic that a configuration (e.g. adapter config) can only endure with the existence of a pipeline config.
One result of the *Event-Storming workshop* is that the pipeline configuration should be decomposed into seperate bounded contexts (configuration items). As a consequence a new relational mapping between the different configs, that spans over multiple microservices, emerged.

Figure 4.10: EERM of configurations



For efficiency reasons the relational semantic between the configs changed. As shown in figure 4.10 a datasource config consists of the scheduling config (1:1 relation) and several pipeline configs, which is composed of the transformation and notification config (1:N relationship). Due to the elimination of the core service a location for the persistence of pipeline properties emerges. The 1:1 relation between the transformation and the pipeline config results in This enables the transformation of imported open data in various representations without importing these datasets for each aggregation (transformation) with subsequent notification handling.

As shown in the Enhanced Relationship-Entity Model (EERM) in figure 4.10 the only *non-weak entity* is the *Datasource config*. This means that all the other configs cannot not exist without it. Due to the distributed architecture almost every weak relationship is modeled on the side of the weak entity resulting in having references to the entity it depends on. For example, the relationship between adapter and pipeline config is modeled with a reference to the datasource config on pipeline config side. As the scheduler

The segregation of datasource properties and pipeline configuration improves efficiency of the system by mapping

The embedded configuration "transformation config" will be assigned to the transformation microservice.

Adapter The adapter bounded context is composed of the adapter configuration database, the adapter service and the scheduler service. The interaction of the user with the system in regards to adapter config creation, modification and deletion accounts for the provision of a REST API instead of an event based message queue.
Two queues have been modeled for the inter-service communication in this context. One queue is responsible for the exchange of the scheduler config (that is part of the adapter config) that is published by the adapter and consumed by the scheduler. Due to the existence of the config REST API on adapter side, this event can either contain the config (fat event) or contain a reference to the config interface (thin event) for its extraction. The queue is configured with a "fixed" routing mechanism (see 2.4.2) to adapt the *Competing Consumer* pattern, resulting in the distribution of the scheduled data import jobs. Increasing the amount of adapter service instances therefore enhances the performance of the system.
The scheduler holds these configurations in its local cache and triggers the adapter service as soon as the scheduled time occurs by sending an event over another queue to the adapter service. The adapter service executes the import of open data from an external source as event processing action and publishes the results to the transformation service via event channel (queue).

Transformation The transformation bounded context is composed of a config database, the transformation and the storage service. It offers, analog to the adapter service, a configuration REST API for CUD operations and receives imported open data for further transformation via event channel and matches its datasource identifier with the transformation configs in the database. If transformation configs that refer to the datasource exist, it will execute the transformation with the subsequent persistence of the data.
As the open data service consumer/consuming application is interested in requesting (un-) aggregated open datasets that will be delivered by the storage

system, a demand emerges for a more efficient solution when scaling large in order to handle enormous amount of requests. The *Command-Query Responsibility Segregation* pattern has been, to some extent, applied to the model for that purpose. The design of the storage model has been split into a “Read“ and a “Write Solution“ (see 2.7).

- **Write Solution** The “Write Solution“ is composed of the transformation service and RabbitMQ. While RabbitMQ acts like a the “Event Bus“ with the provision of a lazy queue [5], the transformation service serves as the write model that receives CUD operations by the previous mentioned reception of “*adapter data events*“ from the adapter service. Optionally an *Event Store* (see 2.7) can be attached to this solution, in order to scale the storage services (and therefore databases) at run time. A storage service instance that will be introduced to the system will be consequently synchronized with the event store in order to maintain consistency.

- **Read Solution** The “Read Solution“ consists of the storage service, that consumes events from the transformation service via “Event Bus“ and provides a REST interface for the request of the data only and a database that serves as “Query Database“ (see 2.7. The routing scheme on the “persistence queue“ (Event Bus) is modeled as an exchange with “fanout“ binding. This means that all events that are published to the exchange will be forwarded to all queues that are bound to the exchange.

  For a better performance the number of databases and storage services can be scaled up. Each service must register a channel to the RabbitMQ broker in order to synchronize with the “Event Store“ to reach the current state and to achieve consistency.

  A “Read“REST interface is exposed to the user (-interface) for querying transformed datasets from the the storage service and therefore from the open data service platform.

Another channel was modeled within this bounded context. It serves the publication of events that indicate that the transformation has been done and persisted.

Notification The notification bounded context consists of a notification service and a configuration database. Likewise the other contexts, it provides a CRUD interface for notification configs. It subscribes to the channel on which events are published by the transformation service to express that the transformed open datasets are available for extraction. In order to evaluate a condition that is provided by the user via config, the transformed data must be delivered to the notification service. This can either be done by embedding the contents of the

[5]Lazy queue persists the events on local storage until consumed

transformation in the event itself (fat event) or by exposing the location of the transformed data for extraction.
With the arrival of one of these events, it matches the pipeline reference of notification configs in the database to the pipeline reference in the event. For each notification config, that refers to the pipeline, a notification is sent to the corresponding platform if the an evaluation of the condition upon the transformed data succeeds.

4.2.3 Choreography

With this architectural modeling approach, the scheduler does not act as an orchestrator, its rather an initiator of a choreographed process flow. The figure 4.11 illustrates the process from the creation of datasource and pipeline configurations, over the extraction of the open data and its subsequent transformation to its persistence and notification to a external platform.
For simplicity reasons the initial state of this process is that all configs have been persisted via REST interface and the *datasource config* is held in the local cache of the scheduler. Only one instance per microservice is modeled in this diagram and all events have been models as *fat events* (see 4.2.2) .
The process flow can be understood as a pipeline in the context of the open data service, as well as a *Saga* in the context of Event-Driven Architecture.

1. **Scheduled time occurs** The scheduler service checks for the schedule properties that are held in its local cache. If a specific time occurs an event is sent to the adapter service to trigger event processing actions (data import).

2. **Data Import** The adpater service searches for the adapter config that corresponds to the event in its config database and executes the data extraction.

3. **Send imported data** After successful import of the data is sent as a *fat event* that contains the data and the internal identified of the datasource to the transformation service.

4. **Transformation** The transformation service receives the imported data with the datasource id and searches in its configuration database for peristed pipeline/transformation configs that are related to the datasource (via id). It executes the function that is contained in the transformation config upon the imported data.

5. **Persist the transformation results** After successful execution of the transformation function the aggregated data will be sent (as fat event) to the storage service.

6. **Data provision** The persisted data will be offered to the user/application for extraction via read-only REST interface by the storage service.
7. **Send notification event** An event is sent to the notification service to indicate that the transformation and its persistence have been completed. The event consists thereby of a pipeline id and the transformed data for condition evaluation upon the data.
8. **Notification dispatch** After the reception of the event from the transformation service, the notification service searches for in the config database for matching notification configs with the pipeline id of the event. For each notification config, the condition will be evaluated and a notification will be sent to the corresponding platform if the condition is met.

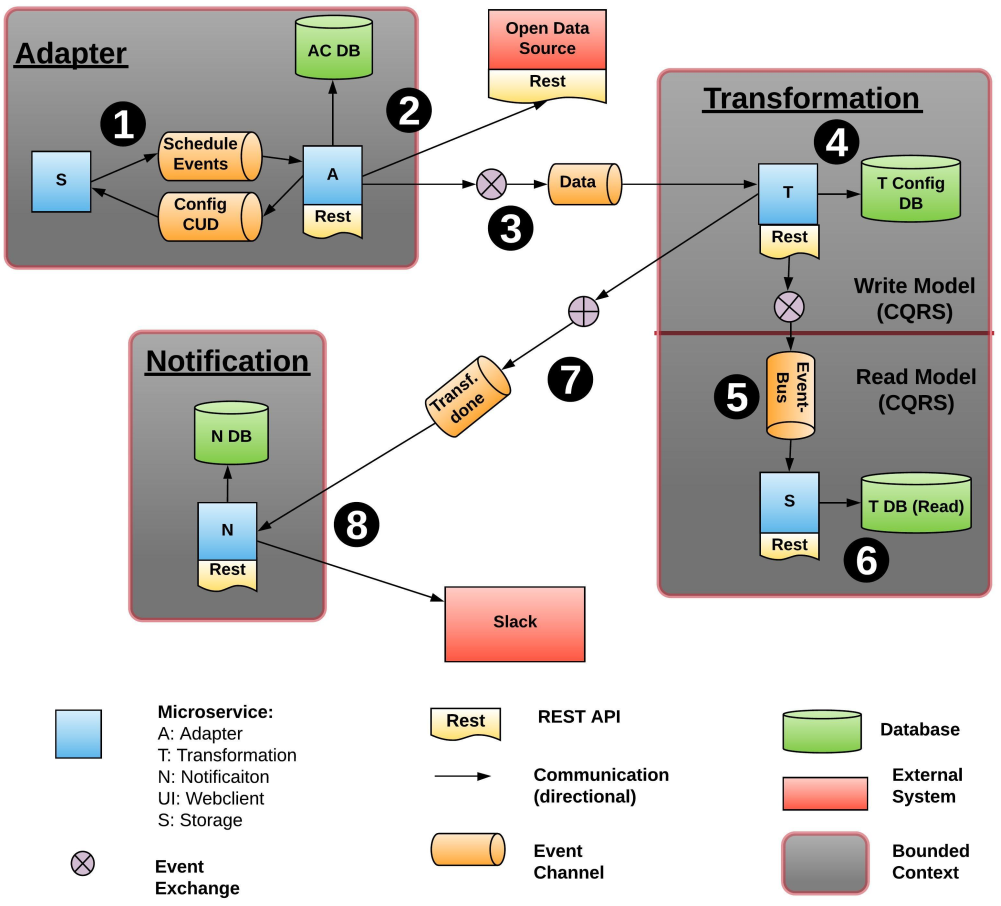


Figure 4.11: Event based choreography of JValue ODS

4.2.4 Evaluation

Microservice and Bounded Context The 1:1 relation constraint between microservice and bounded context could not consistently be applied throughout this modeling approach. In the results of the Event-Storming work, the scheduler was interpreted as an actor in the adapter context (see 4.1.3). In this architectural design, it has been modeled as an own microservice that is located in the bounded context of the adapter. If it is considered to be integrated in the adapter service, scaling would have no effect due to the fact that either all adapter service container instances would be triggered by a scheduled time event or only one if a leader election is implemented.
This applies also for the transformation bounded context due to having the storage service integrated in the context.

Scaling All services in this pattern can be scaled by increasing the amount of instances per service. This does not apply to the scheduler service. If it is scaled with no additional control, each instance would trigger the process flow of the same pipeline. The consideration of keeping it unscaled or to scale it with the leader election pattern emerges (for instance with Apache zookeeper [6]).

Communication analysis Another outcome of this architecture is that the communication throughput can be reduced. This could be achieved on the one hand by separating the configuration databases, resulting in the unnecessity to share the configurations across the microservices. On the other hand by eliminating the orchestrated pattern (the message exchange had to be sent twice, due to the scheduler as intermediate component).

[6]https://zookeeper.apache.org

5 Implementation

The implementation was planned in several consecutive iterations. The strategy pursued is to replace synchronous communication patterns step by step with event based message queues. The major requirement for each iteration is that the system has at least the same functionality as before.
It has been implemented on github [1] as a git fork of the original repository [2]. At the end of all phases integration tests have been modified in order to check the integrity of the system after the implementation. Due to the continuous development of the JValue Open Data Service the collaborators, some developers contributed to the implementation of the architecture.

5.1 Iteration 1: Transformation-Notification segregation

The purpose of this iteration is to segregate the functionality of transforming imported data from the functionality to send notifications to external systems. Both processes are previously defined within the transformation service and are triggered by the scheduler service (see 2.3.3). After a successful transformation and persistence of the imported data the user gets a notification send to an external system (e.g. Slack) if it has been configured in the notification part of the pipeline configuration.
According to the outcomes of the "Event-Storming Workshop" the dispatch of notifications is in another bounded context then the transformation of the imported data and has to be therefore defined in a separated microservice. As a consequence the "notification service" was introduced to the system, whereby it derived all the functionalities of the transformation service in regards of the notification handling.
These modules have been integrated into the notification service:

[1]https://github.com/ke45xumo/open-data-service
[2]https://github.com/jvalue/open-data-service

Notification Dispatch All modules that refer to the transmission of notifications to external systems have been migrated in this iteration.

Notification REST-API The scheduler service triggers the notification emission by sending a HTTP-POST request with the notification configuration as payload to the notification service. All interfaces regarding the notification have been implemented and the scheduler service was modified to use the new endpoint.

Condition evaluation The module for the evaluation of conditions of the transformed data was imported and additionally extended by the functionality to send notifications upon failures in the system. The requirement to keep the functionality of the system was met and even extended.

The technology used for the notification service is NodeJS [3] that runs in a Docker container.

5.2 Iteration 2: Introduction of config databases

Due to the composition of several configuration items within the pipeline config, the need to separate them and assign them to the corresponding bounded contexts emerged (see 4.1.3). To comply with the domain correctness one configuration repository was a applied to each bounded context. This adapts the layered structure of Domain Driven Design. It represents the building block repository and is part of the persistence layer (see 2.1.4).
The technology that is used for the configuration repositories are PostgreSQL [4] databases that introduced to the system as Docker containers. Each microservice communicates with its corresponding configuration database via ODBC.

Adapter configuration Database A Database for the adapter service was created in order to persist the configuration of the properties of a data import. The configuration items, named "Datasource-Configs" contain thereby all the properties, such as the data location, that are mandatory for the extraction of open data from its data sources. The time, as well as the frequency of the extraction is considered to be the scheduler configuration, which has been embedded within the structure of a data source configuration.

Transformation configuration Database The persistence of the properties of a transformation is introduced to the transformation service as a repository. A

[3]https://nodejs.org
[4]https://www.postgresql.org/

conclusion of the Event-Storming workshop was that the core service should be eliminated. As the target database for pipeline configurations vanishes, a need for the persistence of pipeline meta data, which describes the purpose these emerges.

Notification-configuration Database There are currently three types of notification configurations available

- Slack configuration
- Firebase (FCM) configuration
- Webhook configuration

The communication to each platform requires different options (e.g. workspace id for Slack) resulting in the need of differentiating notification configurations into these types. Every notification config refers to a pipeline configuration, enabling a relational mapping of 1:N A database table was created for every type of configuration. The persistence of these configuration types is handled via *Object-relational mapping* (ORM) in typescript [5].

The separation of the entities results in having 1:N relationships between the separated configuration items. For instance can one datasource config be referenced by multiple pipeline configs, which reduces resource consumption by not having to import the same dataset for different transformation results.

Configuration interfaces An interface for the control of configurations has to be implemented for each microservice. Handling these CRUD operations via event communication was taking into consideration, but would interfere with the requirement not to restrict the functionality of the platform. , because the user would no longer be able to persist configs in a convenient way.

As a consequence, an endpoint for the persistence of configuration was implemented for each microservice. The endpoint provides this interface:

- **GET {service url}/config?{query parameter}** This interface returns a list of configurations that match the conditions provided by the query parameters in SJON format.
- **POST {service url}/config** This interface's purpose is to persist the config by sending a configuration in SJON format. It will return the persisted config, containing an id.
- **DELETE {service url}/config/{id}** This interface's purpose is to delete a config. The results of the delete operation can be derived from the status code.

[5]https://github.com/typeorm/typeorm

- **PUT {service url}/config/{id}** This interface's purpose is to update a config with id id by sending a configuration in SJON format. The results of the delete operation can be derived from the status code.

5.3 Iteration 3: Deletion of pipeline repository

Elimination of Core-Service The purpose of the core service, was to persist pipeline configurations. With the successful implementation of the config databases for each service and the corresponding interfaces, the core service is no longer required and has been therefore removed from the system.

UI-Service modification The ui service offers a web based user interface for the creation, deletion and modification of pipeline configurations. Due to the elimination of the pipeline repository, the communication patterns of the ui service have to be modified, in order to move from the communication with the core service to a meaningful way to communicate with the distributed config repositories.

The configuration of a pipeline on this frontend previously consisted of a consecutive selection of properties of its embedded configuration items, such as the notification configuration. A pipeline could only be persisted if all of those steps have been applied and confirmed in the user interface.

This procedure been replaced by the segregation of these embedded items into these single web parts on the webclient:

- **Datasources:** This webpart serves the configuration of the open data import ("Datasource Configs"). The user may select the data location, meta data and the time, as well as the frequency of the data extraction (embedded scheduler config). Data sources are now handled separately. The results will be sent to the adapter service for persistence.
- **Pipelines:** The pipeline configuration on the user interface has been deconstructed and simplified. It only consists of meta data information, the transformation config and a reference to the data source config id.
- **Notifications:** With the decomposition of the pipeline config, After successful pipeline creation the user will be forwarded

Scheduler Service modification For the orchestrated communication with all microservices within the JValue ODS, the scheduler uses pipeline configs in order to dispatch the embedded configs to the corresponding service.
To maintain the logic or respectively the functionality, the requests to the interface of the core service has to be replaced by requests to the microservice that offer the embedded configurations of the pipeline. After the retrieval of the configurations, they have to be composed to the previous pipeline config structure, so that the scheduler can process pipeline executions the same way as before.

5.4 Iteration 4: Transformation-notification communication

Introduction AMQP-Service In this phase a new service is introduced to the current infrastructure of the open data service. A RabbitMQ server embedded in a Docker container has been integrated. It serves as an intermediate layer of communication and provides mechanisms (e.g. channels) for event exchanges between the services.

Notification queue According to the outcomes of the architecture design, a queue (channel) has to be implemented in order to notify the notification service after transformation execution. Several ways to implement this have been evaluated. A trade-off between domain correctness and unnecessary network load in regards of the condition evaluation has been identified.
The condition that are persisted within the notification configuration in the notification config database are evaluated upon the transformed data for the decision whether to dispatch a notification. This can either be achieved by handing the transformed data over to the notification service, where the evaluation takes place, or by sending the conditions to the transformation service for condition evaluation by the transformation service.
The further approach maintains domain correctness by assigning the condition evaluation functionality to the notification service, where it should be semantically located. Due to sending the transformed data from the transformation to the notification service, a higher network load will be the consequence (because the transformed data is in the means larger than the contents of a condition evaluation function that would be sent in the second scenario).
The latter approach causes domain incorrectness by having the transformation service evaluate the conditions for the notification service to check whether a notification should be sent or not, but causes a lower network work load.
The transformation service has in general a higher resource consumption than the notification service because of the execution of the aggregation of data. In order to distribute the load across the microservice evenly and to ensure simpli-

city in development due to domain correctness, the decision has been made to implement the first solution.

Fat vs. thin event Another consideration has been taken into account, either to embed the transformation data in the event body (fat event) or to send a reference to it (thin event), whereas the notification service requests the data synchronously from the storage service. The storage service is expected to have a high network throughput due to having to respond to many data requests by users/applications. The overall performance of the implementation of the fat event is expected to be better and should be therefore chosen.

Notification Message Because of the direct communication between transformation and notification service, the possibility emerged, to send additional meta data, such as the duration of the transformation execution to the notification service. This additional information has been embedded in the message that is sent to the platform, serving the users of the open data service more transparency in regards of when to expect the data after the schedule.

Condition Evaluation The evaluation of condition has been extended with the functionality for error conditions, for the use case that a ODS user wants only to be informed when the data import and aggregation fails. This makes sense, since the import frequency can be set to minutely.
An event with an *undefined* dataset is sent to the notification service, when the transformation of the imported data fails. The user is now able to check whether the pipeline fails, by checking for the condition “data == undefined“ or “data == null“ (in javascript syntax).

At the end of this phase, the orchestrated communication behaviour in regards of the notification has been removed from the scheduler. The requirement to maintain the functionality is given or even extended by the introduction of this feature.

5.5 Iteration 5: Adapter-transformation communication

The communication between adapter and transformation in this phase was implemented by using an event queue. After successful import of the desired data, the adapter service publishes an “AdapterData“ event to this channel.
Also in this section considerations have been made whether to use thin or fat events for the exchange of the imported data. In this case the event has been implemented as a hybrid solution. The event type is defined by these properties:

- **datasourceId** This id is used to search for pipelines or respectively transformation configs with the same datasourceId in order to execute them.
- **data** This is the field where the imported data can be set (when it is used as a fat event)
- **dataLocation** This property can be set to indicate the location (adapter RESTendpoint) where the imported open data sets can be extracted.

If the data is contained in the event itself (" data != null"), the transformation service will process it directly. It extracts otherwise the data from the REST interface referenced by the entry *dataLocataion*.
With the reception of an event the transformation service searches its database for pipeline configs that refer to the "datasourceId" and executes the corresponding transformation functions with the subsequent publication of events to the notification service.

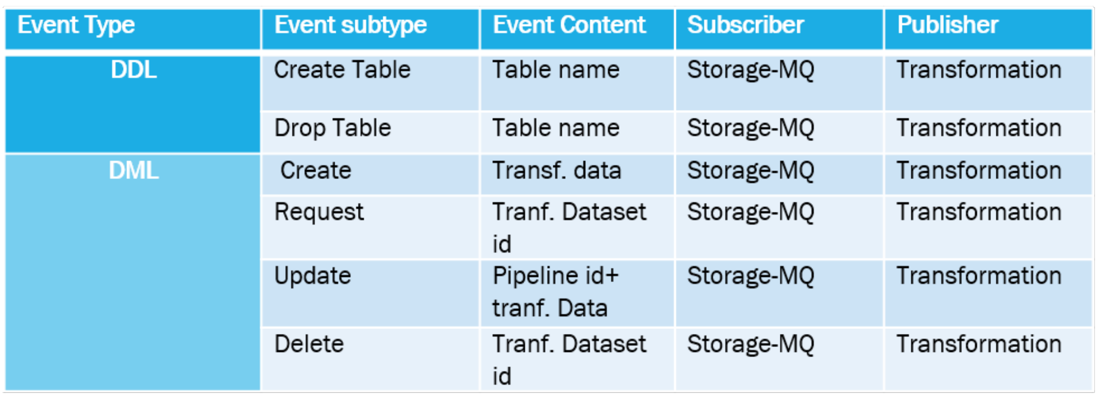
5.6 Iteration 6: Storage-Transformation communication

Introduction of Storage-MQ-Service Hence the previous storage service is based on a predefined docker image that is based on the software PostgREST [6] and is therefore not modifiable, an implementation of a new service that incorporates event based massaging is mandatory.
The technology used for this service is based on NodeJS [7] with typescript [8] framework. It is connected to the storage database via ODBC connection and is subscribed to a queue, which the transformation service publishes various events to. An REST API has been implemented that only accepts HTTP-GET requests to align with the concept of CQRS (see 2.7). The queue thereby acts as an *Event-Bus* between the transformation service and this storage service.

Two event types have been defined with this iteration, whereas these contain event subtypes as shown in figure 5.1. One the one hand, the event type DDL, whereby DDL stands for Data Definition Language. These event types are necessary due to the current structure of the database, where for each pipeline a corresponding table exists. They indicate that a table has to be created upon pipeline creation or that a table has to be deleted within a pipeline deletion. A sub event type for each of these database structure defining operations has been created.

[6]http://postgrest.org
[7]https://nodejs.org
[8]https://www.typescriptlang.org

Figure 5.1: Events of the storage queue



Event Type	Event subtype	Event Content	Subscriber	Publisher
DDL	Create Table	Table name	Storage-MQ	Transformation
	Drop Table	Table name	Storage-MQ	Transformation
DML	Create	Transf. data	Storage-MQ	Transformation
	Request	Tranf. Dataset id	Storage-MQ	Transformation
	Update	Pipeline id+ tranf. Data	Storage-MQ	Transformation
	Delete	Tranf. Dataset id	Storage-MQ	Transformation

On the other hand, the event type DML, which is an acronym for Data Modeling Language is categorized in three further event types that represent CUD operations on the data in the database itself (see figure 5.1).

Modification of transformation service The transformation service has to send an "create table" event to the storage queue when a pipeline config is created by accessing the config REST API. This ensures that the data can be inserted when the transformation is triggered by receiving an event from the "adapter data" queue. Similarly, an "drop table" event will be sent upon the deletion of a pipeline.
After transformation of the imported open data the transformation service sends an "Insert" event (Type: "DML") to the storage queue in oreder to trigger the persistence of the data the new storage service.
The other event subtypes of the "DML" event are currently not used by the open data service.
As final step the communication between the storage and the scheduler has been removed in the scheduler service. Also the transformation communication could be removed because it is no longer needed for persistence purposes.

5.7 Iteration 5: Adapter-scheduler communication

This iteration has not been fully implemented. Nevertheless, the implementation strategy will be demonstrated in this section.
Two queues or respectively channels are introduced in this phase. The objective of both queues is to provide an event based mechanism for the communication between scheduler and adapter service. One queue is responsible for the exchange of “trigger“ events, that are published by the scheduler and consumed by the adapter service to initiate the open data import and therefore the whole process flow of a pipeline. The events thereby contain references (id) to the datasource config, whereas the adapter can identify the corresponding config from its database upon event reception.
The purpose of the other queue is to keep the cached scheduling configs, or respectively adapter configs up to date. These events are published by the adapter service as soon as it receives a CRUD request for datasource config persistence on its REST API. The event types are thereby categorized in CRUD operations, which illustrate a change of the datasource config to the scheduler.

6 Evaluation

6.1 Requirements

6.1.1 Functional

F1 Choreography instead of orchestration The requirement was to move from a orchestrated communication pattern to a event-based choreography. This could be achieved by removing the blocking REST API requests by the scheduler and introducing queues between the microservices for information or respectively event exchange. The scheduler is no longer assigned to the role as an orchestra tor. It can be moreover understood as an "event flow initiator", since it publishes the initial event to the system, which engages other services to publish events as well.

F2 Asynchronous messaging The goal of this architecture was to eliminate all synchronous communication mechanisms (request reply). This has only been implemented to some extent. For some endpoints, it was either not applicable, for instance ODBC connections to the databases or it was unreasonable to do so (e.g. for the REST interface that are exposed to the user/application.

F3 Inter-service dependencies This requirement states that a microservice should not communicate with other microservices directly. It should instead use the event bus, or respectively the channels for that purpose. This has been applied to some microservices, mainly those who consume and publish *fat events*. The microservices that use thin events, like the scheduler still depend on communicating directly with the adapter service for datasource config extraction.

6.1.2 Non-functional

N1 Functionality of the system has to be preserved The demand to the system is that the functionality of the ODS platform stay the same. This could be fully achieved. The system was even extend by further functional aspects, such as the provision of conditional evaluation of platform failures.

N2 Increase in performance The overall performance should be increased (untested) due to the fact that less communication takes place within the system. The reason for this is the elimination of the orchestrated behaviour of the scheduler, that forced inter-service communication to be exchanged at least twice. The notification execution, for instance, required the scheduler to extract transformed data from the transformation service and then pass it to this service. This problem has been resolved by the direct communication in an choreographed manner.

Another aspect that should be considered in terms of performance increase, is that pipelines that refer to the same datasource, will not instruct the adapter service to extract the data from the open data sources anymore. Due to the separation of the context of pipeline and datasource and their corresponding configs, the data is only extracted once and processed several times.

N3 Collaboration and Development This non-functional requirement is about how to handle the development for a successful integration of the code into the existing repository, as well about collaboration with the open data contributor team.
The development was planned in several phases, that each did not reduce functionality and were applicable to the repository. The implementation of these phases was not done as planned. The challenge of implementing simultaneously on several, different microservices lead to the tendency to implement on one microservice one after another, or to implement similar functionalities within an iteration. As a consequence the git “pull requests“ that have been made for the git “master branch“ were not properly separated and lead to difficulties in terms of integration.
In regards to the collaboration, every major change that has been made was discussed in a collaboration standup meeting, that was scheduled every two weeks.

6.2 Scalability

The architecture was planned with the intention to improve the performance of the system. Scalability plays thereby an important role and was consequently included in the modeling process. The implementation build a foundation for the system to scale large but did not specifically setup scaling rules or similar functionality, like the configuration of Kubernetes [1].

6.3 Further improvements

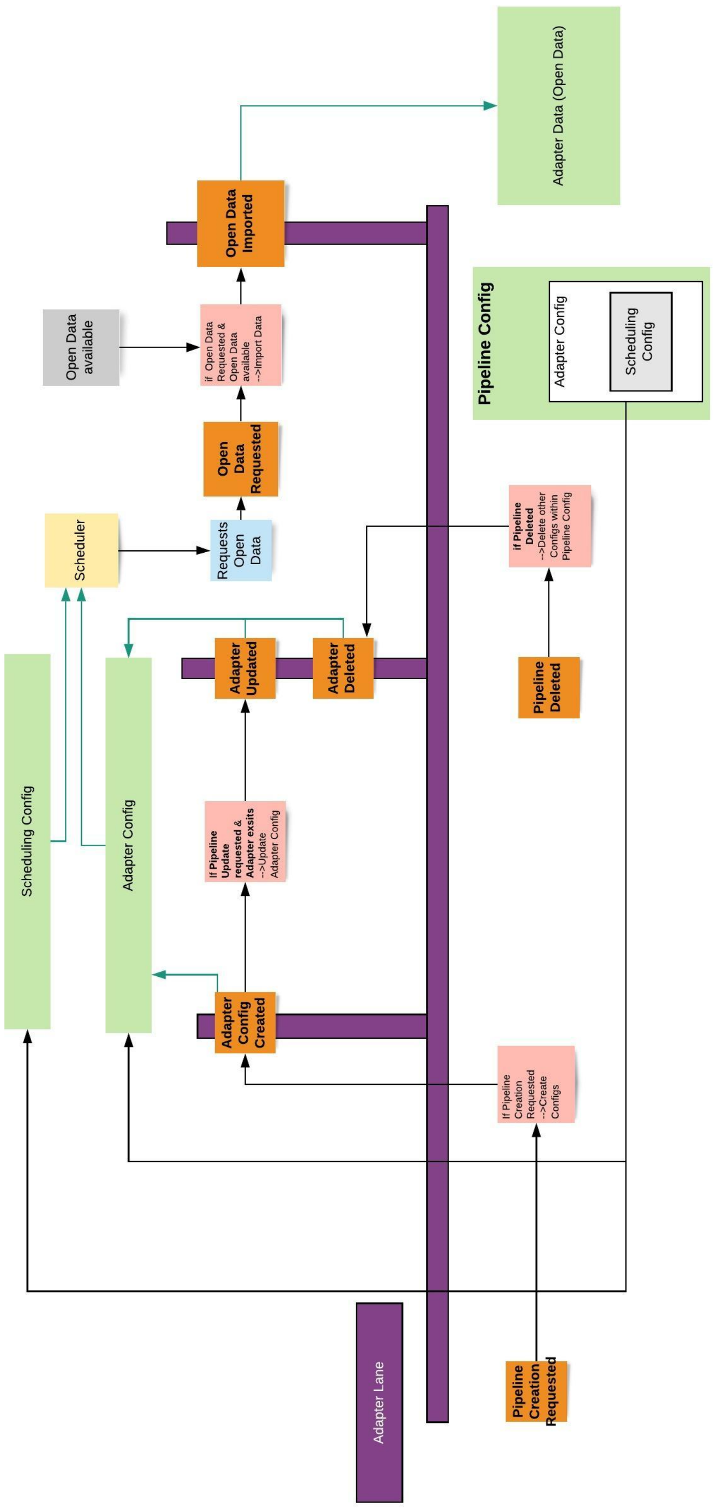
The goal of this thesis was to achieve choreography within the architecture of the JValue Open Data Service, by applying event driven architecture. A further improvement to the system in regards of event based communication could be to extend the inbound and outbound Application Programming Interfaces with event queues. The implementation of an outbound queue can enable users or respectively applications to consume transformed open data from the platform. The introduction of an inbound event channel could extend the functionality by enabling the system to import or respectively consume from open data message queues.

[1]https://kubernetes.io/

Appendix A Event storming workshop

Appendix B Event driven architecture - Scaled

Figure 6.1: Results of the event storming workshop - Adapter Lane



Adapter Lane
Pipeline Creation Requested
If Pipeline Creation Requested -->Create Configs
Adapter Config Created
If Pipeline Update requested & Adapter exsits -->Update Adapter Config
Adapter Updated
Adapter Deleted
Pipeline Deleted
If Pipeline Deleted -->Delete other Configs within Pipeline Config
Scheduling Config
Adapter Config
Scheduler
Requests Open Data
Open Data Requested
Open Data available
if Open Data Requested & Open Data available -->Import Data
Open Data Imported
Adapter Data (Open Data)
Pipeline Config
Adapter Config
Scheduling Config

Figure 6.2: Results of the event storming workshop - Notification Lane

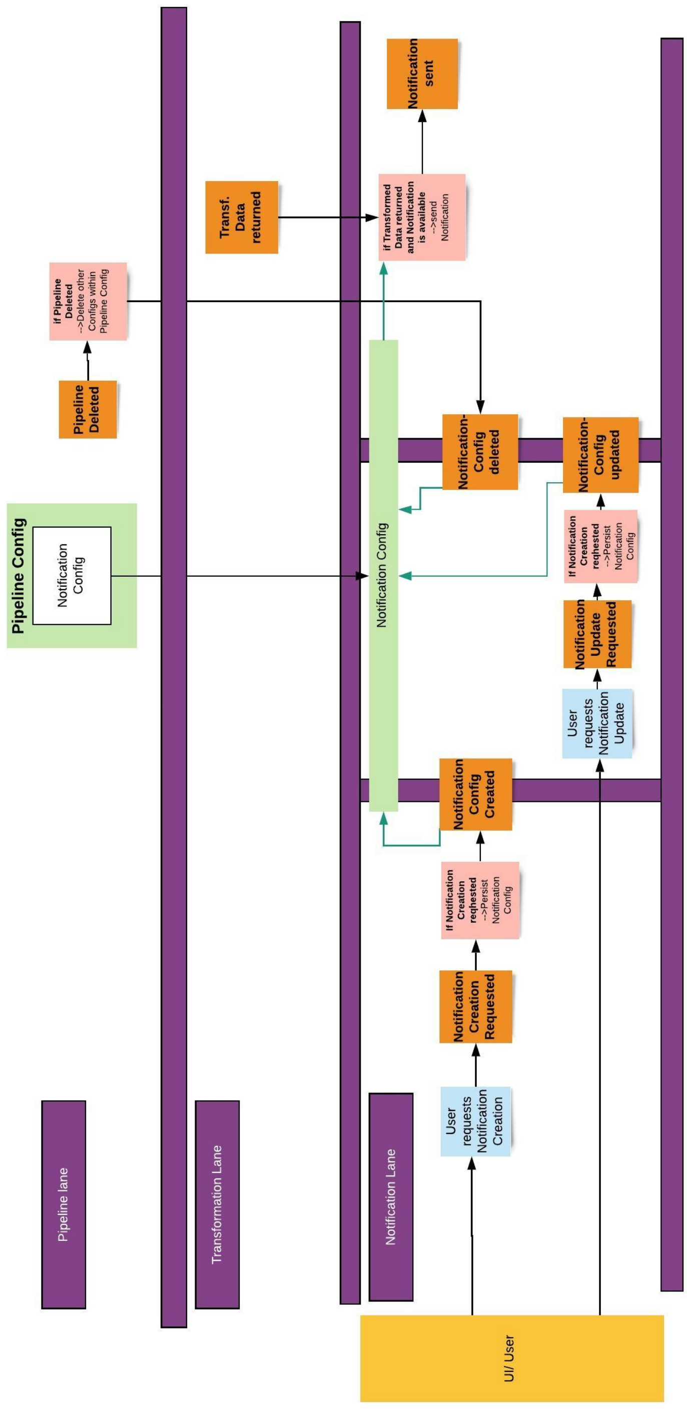
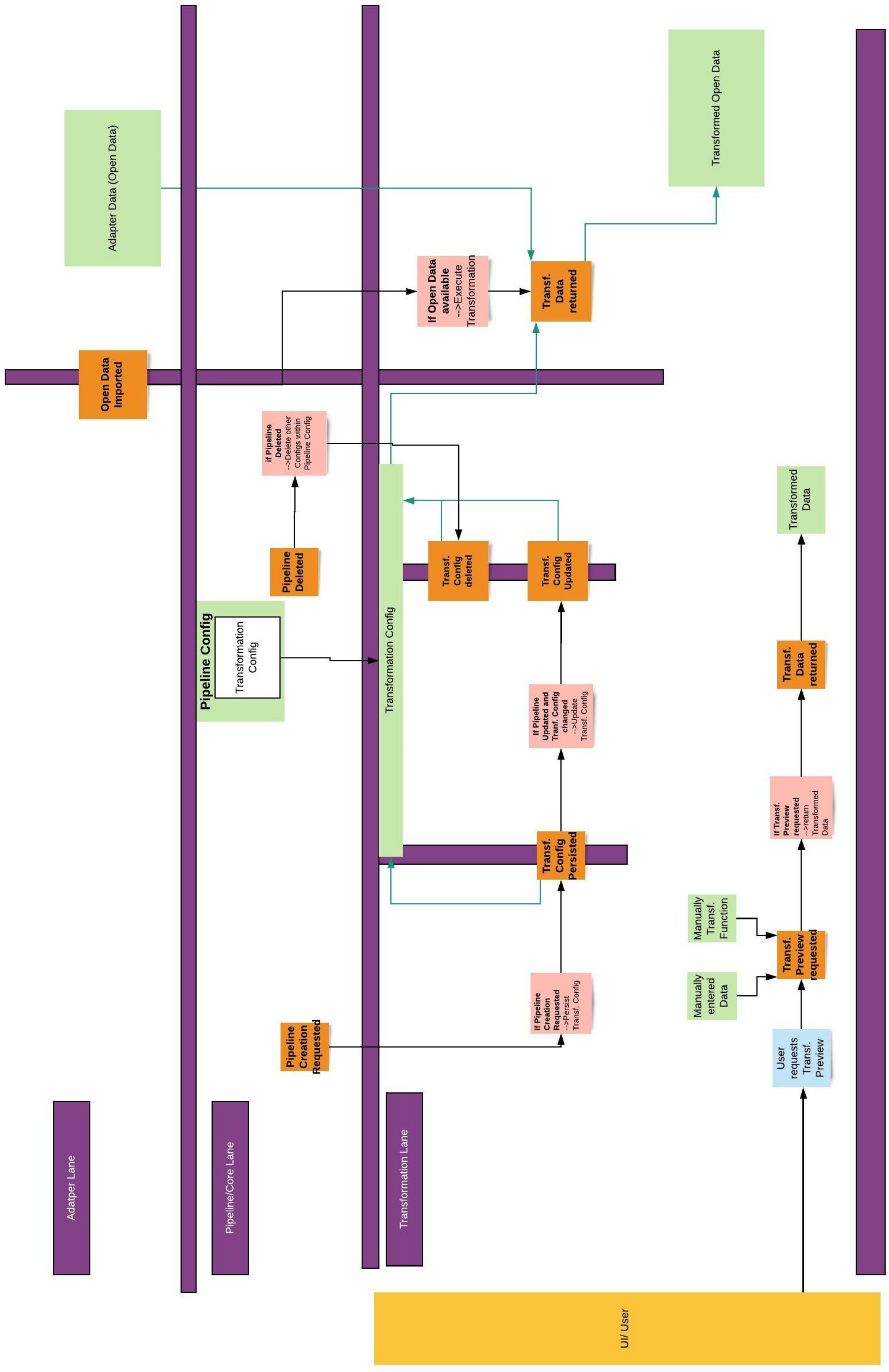

Pipeline lane
Transformation Lane
Notification Lane
UI/ User
User requests Notification Creation
Notification Creation Requested
If Notification Creation reqhested -->Persist Notification Config
Notification Config Created
User requests Notification Update
Notification Update Requested
If Notification Creation reqhested -->Persist Notification Config
Notification-Config updated
Notification-Config deleted
Notification Config
Pipeline Config
Notification Config
Pipeline Deleted
If Pipeline Deleted -->Delete other Configs within Pipeline Config
Transf. Data returned
If Transformed Data returned and Notification is available -->send Notification
Notification sent

Figure 6.3: Results of the event storming workshop - Transformation Lane



Adatper Lane
Pipeline/Core Lane
Transformation Lane
UI/ User
Adapter Data (Open Data)
Open Data Imported
If Open Data available -->Execute Transformation
Transf. Data returned
Transformed Open Data
If Pipeline Deleted -->Delete other Configs within Pipeline Config
Pipeline Deleted
Pipeline Config
Transformation Config
Transformation Config
Transf. Config deleted
Transf. Config Updated
If Pipeline Updated and Tranf. Config changed -->Update Transf. Config
Transf. Config Persisted
If Pipeline Creation Requested -->Persist Transf. Config
Pipeline Creation Requested
Manually Transf. Function
Manually entered Data
Transf. Preview requested
User requests Transf. Preview
If Transf. Preview requested -->return Transformed Data
Transf. Data returned
Transformed Data

Figure 6.4: Results of the event storming workshop - Storage Lane

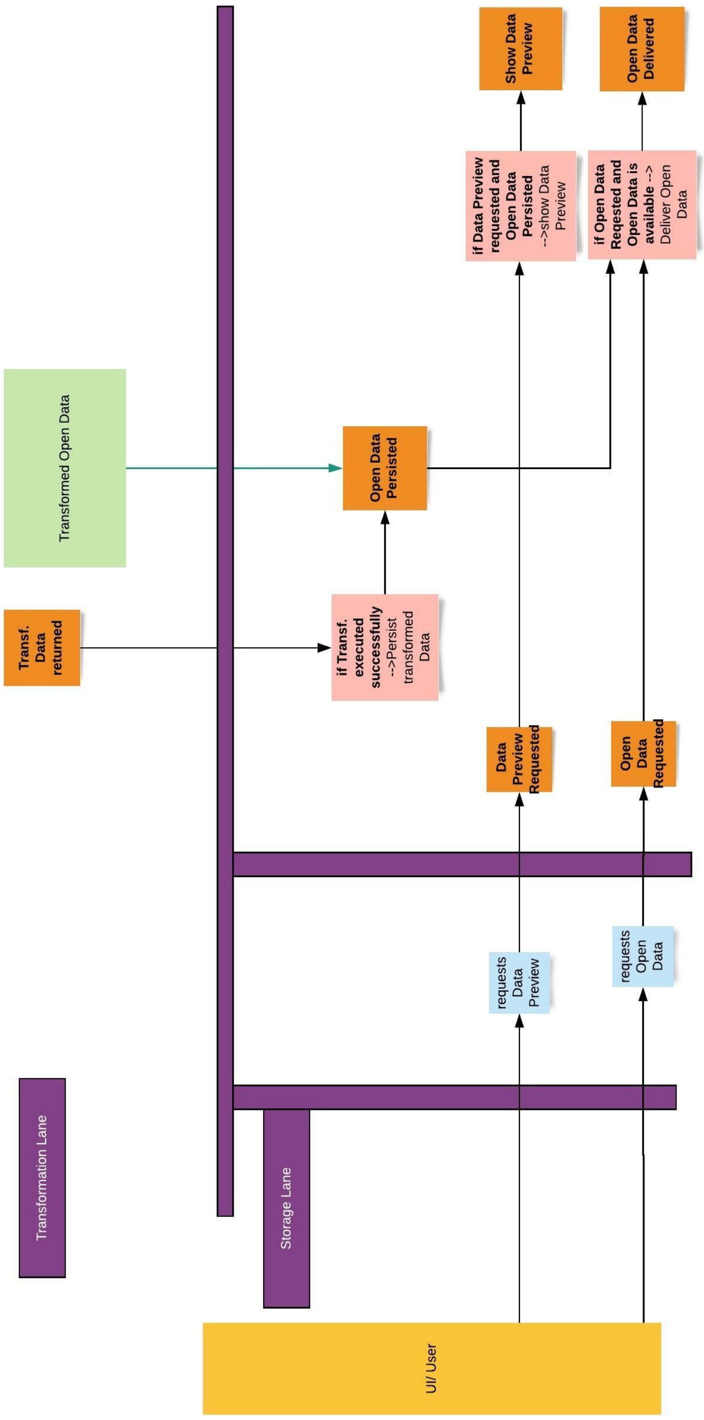
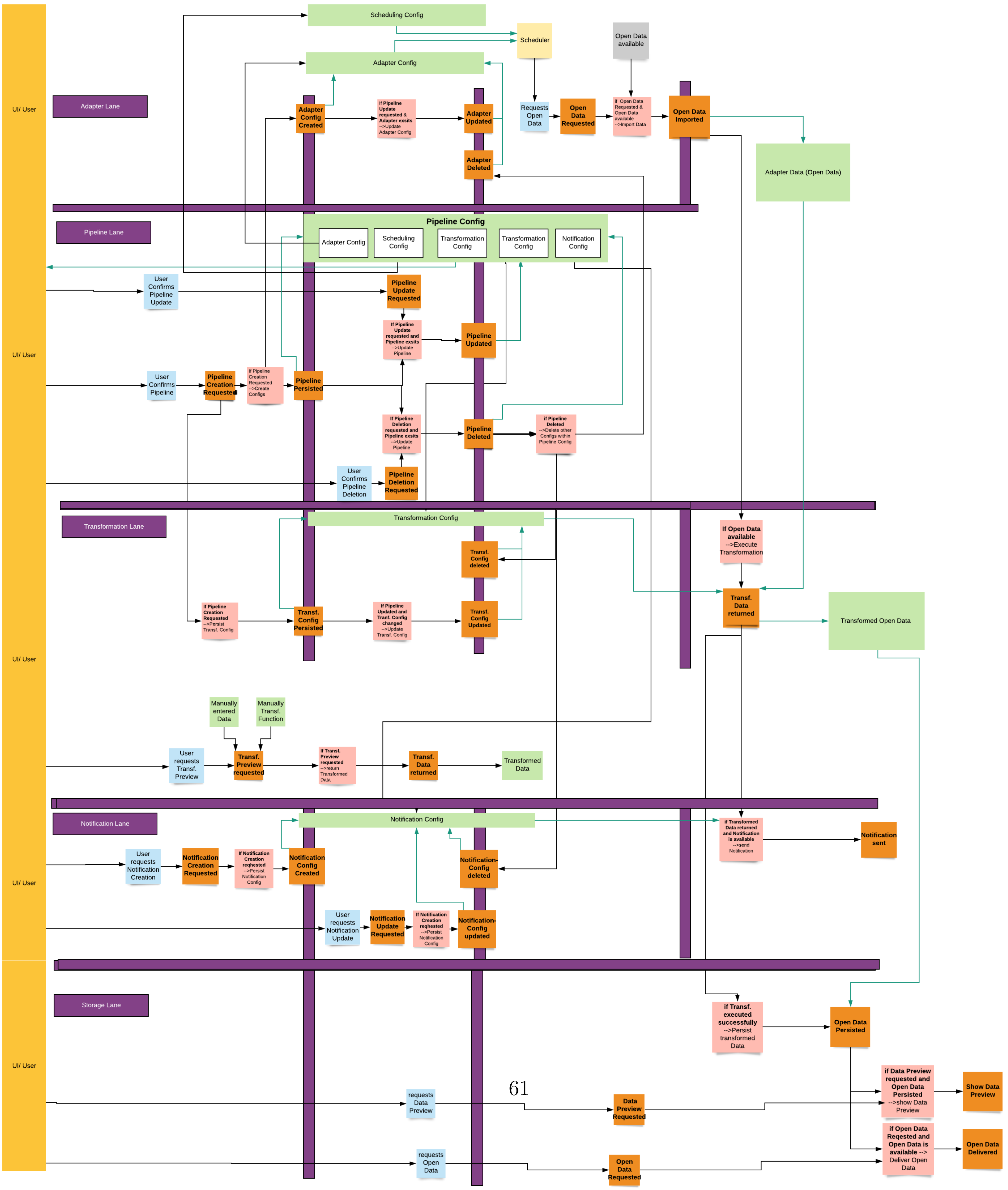
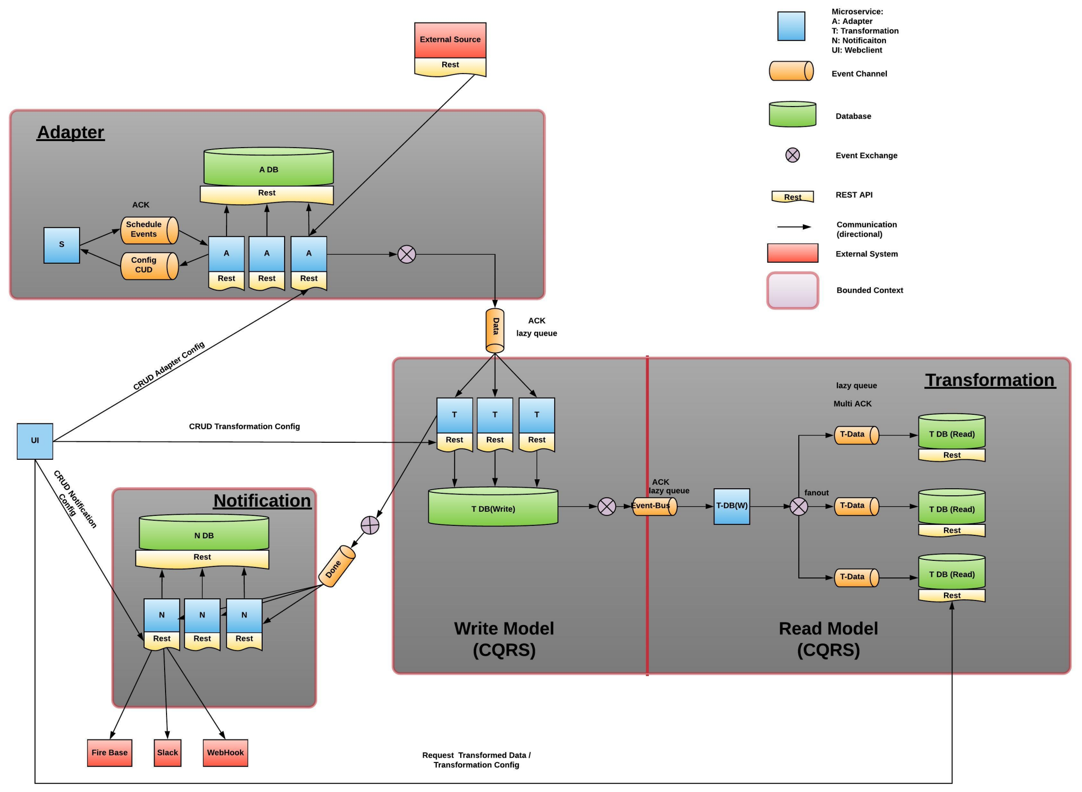


Figure 6.5: Results of the event storming workshop - Overview



UI/ User
Adapter Lane
Scheduling Config
Scheduler
Open Data available
Adapter Config
Adapter Config Created
If Pipeline Update requested & Adapter exsits -->Update Adapter Config
Adapter Updated
Adapter Deleted
Requests Open Data
Open Data Requested
if Open Data Requested & Open Data available -->Import Data
Open Data Imported
Adapter Data (Open Data)
Pipeline Lane
Pipeline Config
Adapter Config
Scheduling Config
Transformation Config
Transformation Config
Notification Config
User Confirms Pipeline Update
Pipeline Update Requested
If Pipeline Update requested and Pipeline exsits -->Update Pipeline
Pipeline Updated
User Confirms Pipeline
Pipeline Creation Requested
If Pipeline Creation Requested -->Create Configs
Pipeline Persisted
If Pipeline Deletion requested and Pipeline exsits -->Update Pipeline
Pipeline Deleted
if Pipeline Deleted -->Delete other Configs within Pipeline Config
User Confirms Pipeline Deletion
Pipeline Deletion Requested
Transformation Lane
Transformation Config
If Open Data available -->Execute Transformation
Transf. Config deleted
Transf. Data returned
Transformed Open Data
If Pipeline Creation Requested -->Persist Transf. Config
Transf. Config Persisted
If Pipeline Updated and Tranf. Config changed -->Update Transf. Config
Transf. Config Updated
Manually entered Data
Manually Transf. Function
User requests Transf. Preview
Transf. Preview requested
If Transf. Preview requested -->return Transformed Data
Transf. Data returned
Transformed Data
Notification Lane
Notification Config
If Transformed Data returned and Notification is available -->send Notification
Notification sent
User requests Notification Creation
Notification Creation Requested
If Notification Creation reqhested -->Persist Notification Config
Notification Created
Notification-Config deleted
User requests Notification Update
Notification Update Requested
If Notification Creation reqhested -->Persist Notification Config
Notification-Config updated
Storage Lane
if Transf. executed successfully -->Persist transformed Data
Open Data Persisted
if Data Preview requested and Open Data Persisted -->show Data Preview
Show Data Preview
requests Data Preview
Data Preview Requested
if Open Data Reqested and Open Data is available --> Deliver Open Data
Open Data Delivered
requests Open Data
Open Data Requested

Figure 6.6: Event Driven Architecture of ODS



References

[al20a] Jerome Boyer et al. *Command-Query Responsibility Segregation (CQRS).* 2020. URL: https://ibm-cloud-architecture.github.io/refarch-eda/patterns/cqrs/ (visited on 12/05/2020).

[al20b] Jerome Boyer et al. *Event driven solution implementation methodology.* 2020. URL: https://ibm-cloud-architecture.github.io/refarch-eda/methodology/event-storming/ (visited on 03/05/2020).

[Avr07] Abel Avram. *Domain-Driven Design Quickly.* Lulu.com, 2007. ISBN: 1411609255. URL: https://www.infoq.com/minibooks/domain-driven-design-quickly.

[BK04] David Burdett and Nickolas Kavantzas. *WS Choreography Model Overview.* WD not longer in development. http://www.w3.org/TR/2004/WD-ws-chor-model-20040324/. W3C, Mar. 2004.

[Bra19] Alberto Brandolini. *Introducing EventStormin.* Apress, 2019. URL: https://leanpub.com/introducing_eventstorming.

[EN10] Opher Etzion and Peter Niblett. *Event Processing in Action.* 1st. USA: Manning Publications Co., 2010. ISBN: 1935182218.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[Fai11] Ted Faison. *Event-Based Programming: Taking Events to the Limit.* 1st. USA: Apress, 2011. ISBN: 1430243260.

[FF06] Martin Fowler and Matthew Foemmel. *Continuous integration.* 2006. URL: https://martinfowler.com/articles/continuousIntegration.html.

[HPX13] Sven Helmer, Alexandra Poulovassilis and Fatos Xhafa. *Reasoning in Event-Based Distributed Systems.* Springer Publishing Company, Incorporated, 2013. ISBN: 3642267866.

[HW12] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2012. ISBN: 9780133065107. URL: https://books.google.de/books?id=qqB7nrrna%5C_sC.

[MFP06] Gero Mühl, Ludger Fiege and Peter Pietzuch. *Distributed Event-Based Systems.* Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 3540326510.

[MT15] S. Millett and N. Tune. *Patterns, Principles, and Practices of Domain-Driven Design*. Wiley, 2015. ISBN: 9781118714706. URL: https://books.google.de/books?id=SsK5mwEACAAJ.

[Nad+16] Irakli Nadareishvili et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. 1st. O'Reilly Media, Inc., 2016. ISBN: 1491956259.

[New15] S. Newman. *Building Microservices*. O'Reilly Media, 2015. ISBN: 9781491950357. URL: https://books.google.de/books?id=1uUDoQEACAAJ.

[Ric15] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015. ISBN: 9781491925409.

[Ric20a] Chris Richardson. *Pattern: Event sourcing*. 2020. URL: https://microservices.io/patterns/data/event-sourcing.html (visited on 10/04/2020).

[Ric20b] Chris Richardson. *Pattern: Saga*. 2020. URL: https://microservices.io/patterns/data/saga.html (visited on 12/04/2020).

[Ver13] Vaughn Vernon. *Implementing Domain-Driven Design*. 1st. Addison-Wesley Professional, 2013. ISBN: 0321834577.

[Wol16] E. Wolff. *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform, 2016. ISBN: 9781523361250. URL: https://books.google.de/books?id=X7YzjwEACAAJ.