FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Teemu Varsala

MODERNIZATION OF A LEGACY SYSTEM: EVENT STREAMING WITH APACHE KAFKA AND SPRING BOOT

Master’s Thesis
Degree Programme in Computer Science and Engineering
January 2023

Varsala T. (2023) Modernization of a Legacy System: Event Streaming with Apache Kafka and Spring Boot. University of Oulu, Degree Programme in Computer Science and Engineering, 77 p.

ABSTRACT

In this thesis, we will design, implement, and evaluate a brand new replacement, the Watcher, for a legacy system built over two decades ago. The Watcher is able to track changes in our PDM system, and notify users of the changes by email or as a push notification using SSE. Functional requirements for the new system come from the legacy system including the possibility to create subscriptions with a wide range of options to filter out redundant data traffic. The Watcher will also be able to carry out all operations than the predecessor with increased performance and efficiency. The main focus is on scalability, maintainability, and fault tolerance. The reason for building a new system is mainly the cost of maintainability and further development of the legacy system as well as features removed due to obsolete technologies.

In the literature review, we go through the theory of the technologies related to the project. We create a REST API with Spring Boot for interactions between users and the system, implement powerful event streaming and processing environment using Apache Kafka, and build a message service responsible for providing information via scheduled emails or SSE. In the end, we will use Docker to containerize all the services.

In the project design, we present functional as well as technical requirements that we use later on to evaluate the project's success. We also compare the legacy system to the new one using metrics such as speed and ease of the installation process. In the end, we discuss the project's future including steps before going to production such as automatic testing, and further development for years to come such as orchestration.

Keywords: Java Spring Boot, Kafka, event-driven architecture, streaming

Varsala T. (2023) Legacy palvelun uudistaminen: reaaliaikajärjestelmä Apache Kafkaa ja Spring Bootia hyödyntäen. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 77 s.

TIIVISTELMÄ

Tässä työssä suunnittelemme, toteutamme ja arvioimme uuden järjestelmän, joka tulee korvaamaan yli kaksi vuosikymmentä sitten luodun legacy-järjestelmän. Tämä uusi järjestelmä, "the Watcher", kykenee seuraamaan muutoksia meidän PDM järjestelmässämme, ja ilmoittamaan muutoksista käyttäjille sähköpostilla, sekä push-ilmoituksilla. Hyödynnämme tässä työssä toiminnallisia vaatimuksia, jotka ovat määritelty jo vanhalle järjestelmälle. Esimerkiksi tilausten luominen käyttäen useita suodattimia vähentäen samalla tarpeetonta dataliikennettä. "The Watcher"kykenee suoriutumaan kaikista tehtävistä joista vanha järjestelmäkin, sekä lisäominaisuutena se tarjoaa paremman suorituskyvyn, sekä tehokkuuden. Pääpainona järjestelmässä on skaalautuvuus, ylläpidettävyys ja vikasietoisuus.

Kirjallisuuskatsauksessa käymme läpi projektiin liittyvien teknologioiden teorian. Toteutamme Spring Boot ohjelmointikehyksen avulla REST-rajapinnan, jonka välityksellä käyttäjät voivat kommunikoida järjestelmän kanssa. Rakennamme myös tehokkaan ympäristön datan käsittelyyn ja reaaliaikaiseen viestintään käyttäen Apache Kafkaa. Viimeiseksi luomme viestipalvelun, joka vastaa käyttäjien informoimisesta hyödyntäen SSE:ksi kutsuttua teknologiaa, sekä lähettämällä sähköpostiviestejä käyttäjien toivomana ajankohtana. Lopuksi vielä sijoitamme kaikki palvelut kontteihin Dockerin avulla.

Projektin suunnitteluosiossa esittelemme niin toiminnalliset, kuin teknisetkin vaatimukset, joiden avulla arvioimme myöhemmin projektin onnistumista. Vertaamme myös vanhaa ja uutta järjestelmää käyttäen metriikoita kuten nopeus ja asennusprosessin yksinkertaisuus. Lopussa keskustelemme projektin tulevaisuudesta sisältäen vaiheet jotka tulisi suorittaa ennen kuin järjestelmä voidaan ottaa tuotantokäyttöön kuten automaattinen testaus, sekä toiminnallisuuksien kehitys tulevina vuosina.

Avainsanat: Java Spring Boot, Kafka, event-driven architecture, event streaming

TABLE OF CONTENTS

FOREWORD

This thesis was written for my Master's degree in Computer Science and Engineering in cooperation with Roima Intelligence. The topic is significant due to the vast amount of legacy systems while technologies in information systems are developing rapidly.

First and foremost, I wish to thank my fiancée for taking care of our little kids during the day and at night, as well as for continuous support and tolerance of my absence. Also, I would like to thank Roima Intelligence for giving me this great opportunity to turn one system into its new era, as well as my co-workers Rami Norolahti, Minna Kalli, Jesse Urholin, and Antti Lehesvuori for valuable guidance. I would also like to thank Dr. Aku Visuri, the supervisor of this thesis, for the considerable effort during the process and for the excellent counseling. M.Sc. Iván Sánchez Milara deserves acknowledgment for his work as the second examiner.

Oulu, January 27th, 2023

Teemu Varsala

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
CQRS	Command Query Responsibility Segregation
DB	Database
DSL	Domain-Specific Language
EDS	Event-Driven System
FR	Functional Requirement
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LS	Legacy System
LSS	Legacy Subscription Service
OCI	Open Container Initiative
OS	Operating System
PDM	Product Data Management
REST	Representational State Transfer
SSE	Server-Sent Events
SQL	Structured Query Language
SRP	Single Responsibility Principle
TR	Technical Requirement
URI	Uniform Resource Identifier
VM	Virtual Machine
VS Code	Visual Studio Code

1. INTRODUCTION

The world is changing, and the demand for more agile solutions in information technology is increasing. The vast amount of systems created decades ago are left in a state where they cannot keep up with the requirements of modern systems. Even though legacy systems had the latest technologies and features at the time, however, nowadays they are considered inflexible with high maintainability costs and poor scalability. Legacy systems also lack sufficient expertise, resulting in the system eventually becoming outdated. Furthermore, novel and better technologies are being developed rapidly to replace older ones. Therefore, general improvements for legacy system often targets to improve maintainability and scalability, which leads lower overall costs.

In this thesis, we will design, implement, and evaluate a brand new replacement, the Watcher, for a legacy system built over two decades ago. The Watcher is able to track changes in our PDM system, and notify users automatically of the changes by email or as a push notification using SSE. The changes can be document updates, item status changes, and almost all modifications in the PDM system we can think of. The purpose of PDM systems is to facilitate product data management of companies by offering variety of functionality to order data in the desired way, search data, automate processes, and keep history of the data. Thus PDM systems are able to decrease the amount of time consuming routine tasks such as ordering and categorizing files constantly by managing these files for us. PDM systems may also restrict editing to only one user at a time preventing any conflicts. Several companies have their own PDM system implementations, for example SolidWorks PDM[1] and Siemens Teamcenter PDM/PLM[2].

Functional requirements for the new system comes from the legacy system including possibility to create subscriptions with a wide range of options to filter out redundant data traffic. The Watcher will also be able to carry out all operations than the predecessor with increased performance and efficiency. The main focus is on scalability, maintainability, and fault tolerance. The reason for building a new system is mainly the cost of maintainability and further development of the legacy system as well as features removed due to obsolete technologies.

Legacy systems often suffer from similar challenges, which could be solved partially by changing traditional request-response model and monolith infrastructure. Therefore we update the legacy system by building an event-driven architecture consisting of several smaller services, and thus separate responsibilities.

[1] https://www.solidworks.com/product/solidworks-pdm

[2] https://www.plm.automation.siemens.com/global/en/products/teamcenter/

The starting point for the project was to find a solution to get data out from the database of the PDM system, and bring it into the Watcher. Event streaming is relatively new, but widely used, thus we decided to look for the answer in that direction. Also, since the main reasons for replacing the legacy system is poor maintainability and outdated technologies we chose to construct the system from several smaller services having a single responsibility and to use popular technologies with long term support and a vast online community. In the end, we expect to have a system capable of tracking changes that happen in the database of the PDM system. Also, users need to be able to interact with the system in order to determine what types of changes in the PDM system they want be notified of. Finally, we need to figure out a way to match those changes with the ones that users have specified and deliver them by email or as a push notification. Even though the functionality is most likely very application-specific, we aim to find a general-purpose solution for the architecture that can be applied to many different use cases.

During this project, we create a REST API with Spring Boot for interactions between users and the system, implement powerful event streaming and processing environment using Apache Kafka, and build a message service responsible for providing the information via scheduled emails or SSE. In the end, we will use Docker to containerize all the services. The technology stack we came up with was the result of research how to do event streaming, process data, track changes in a database, and just company or personal preferences.

The rest of the thesis is structured as follows. In chapter 2, related work, we go through the theory of the technologies related to the project. The design contains the initial plan of how we approached this challenge in terms of architecture as well as definitions for the functional and technical requirements. The design is based on the knowledge gained during the research for related work. The implementation stage provides a detailed but straightforward review of the process we went through to put everything together. After implementation, we compare the requirements to the achieved results as well as the legacy system to the new system. In chapter 6, discussion and evaluation, we analyze all prior phases, design, implementation, and results, after which we will conclude all in chapter 7.

2. EVENT-DRIVEN SYSTEMS AND PDM

This chapter provides a comprehensive overview of the background work of this project. We first describe the environment into which the system built during this project will be integrated. Other concepts, such as design patterns and cutting-edge technologies, will also be covered later.

Technologies presented in this chapter are utilized in order to create a change tracking system that allows us to specify what type of information we want to stream from the PDM system. The architecture can be divided in three main components: user interaction, architecture and containers, and data stream from the PDM system and data processing. In section REST 2.2, we explain the technology how users are able to interact with the system. Then we go through technologies how all the services are able to work together in sections Docker 2.3 and System Architecture 2.4. Finally, in section Kafka Ecosystem 2.6 we explain the technologies that are required in order to achieve a constant data stream between the PDM system and the system we build in this project.

2.1. Product Data Management

Product Data Management (PDM) is gaining popularity, which can be seen in the constant increase in the number of conferences and companies offering PDM solutions. Although PDM encompasses more than just software [1], this project focuses purely on the technical side, on a system that further enhances the use of PDM.

Product data or product information is simply all the data related to the product. To name a few; 3D models, brochures, orders, invoices, and part lists can all be subjects of product information. Many PDM systems are developed for product design rather than for handling delivery and order processes. PDM systems vary significantly between different vendors, and terms are not standardized. However, a couple of general elements exist and can be found in many of these systems. [1]

The list below covers common elements that can be found in most of the PDM systems. [1]. These elements describe how product data can be managed.

1. **Item management**: For example, some objects, such as a 3D model of a car door, can be managed by a PDM system. Furthermore, the car door may probably have other information, such as version and category linked to it.

2. **Document management**: Documents are a subclass of items. Thus, they have all the item properties in addition to the document-specific properties.

3. **Product structure management**: It is common for a product to have some sort of hierarchical structure, which consists of other smaller products, and even they consist of other smaller products, and so on. All these smaller products are components of something bigger.

4. **Change management**: Items and documents often have relations to other items and documents. For a successful PDM system, it is crucial to have support for changed information.

Due to commercial pressure, vendors are forced to discover new terms that describe more or less the same concept. Although these terms cover essentially the same concept of PDM, they all emphasize different characteristics of PDM. [1]

2.2. REST

The default protocol for building services before REST was the Simple Object Access Protocol, better known as SOAP, which gained massive popularity among developers. However, the REST became the top choice by being more lightweight, straightforward and supporting multiple data formats. [2]

REST is an architectural pattern for implementing web interfaces. It consists of guidelines that must be followed to satisfy the REST constraints described in listing 2.2.[3] Although Leonard Richardson developed a model of different maturity levels after the REST was defined. The model comprises four levels in total, and the highest one is Hypermedia Controls.[4]

Roy T. Fielding, in his doctoral dissertation in 2000, describes the principles of REST. [3]:

1. **Client - Server**: Data storage separated from the user interface facilitates the portability of systems and improves the scalability of these independent components. This separation of concerns helps manage roles for the client and server as well.

2. **Stateless**: Stateless constraint imposes that each request a client makes to a server has to contain enough information to carry out the request without using any stored data on the server side. This constraint comes with various advantages, including scalability, visibility, and reliability. As a disadvantage, repetitive calls might load the network more than a stateful model.

3. **Cache**: Even though the constraint described above may limit the network performance, caching, on the other hand, decreases the traffic and thus increases the overall performance. Although, if data

alters in the fast phase, cached data will only remain valid for a short period of time, leading to a situation where the cache needs to be refreshed in the rapid phase as well.

4. **Layered system**: A layered system provides abstraction between clients and servers. Clients do not know anything about the layers after the one they are in direct contact with. [3] Load-balancing is a concept that utilizes layering components by having, for example, some sort of proxy or controller standing between clients and servers. Clients interact with the proxy, and the proxy interacts with the server. In the case of modern web services, we are talking about a complex network comprising multiple servers with multiple applications and application instances.[5]

5. **Uniform interface**: REST emphasizes a Uniform interface defining four interface constraints: "identification of resources; manipulation of resources through representations; self-descriptive messages; and hypermedia as the engine of application state." A uniform interface states that one Uniform Resource Identifier belongs to one resource only. Since the interface in REST is highly standardized, it may cost some efficiency depending on the needs of an application.

6. **Code-On-Demand**: The only optional constraint for REST since it reduces client visibility. Code-On-Demand extends functionality on the client side in the form of transferable executable code.

2.3. Docker

Docker is an operating system (OS) level virtualization platform for running applications in a loosely isolated environment called a container. Originally Docker was created by dotCloud inc, which changed its name to Docker later on, and decided to put their focus on the development of Docker product. In the early days of Docker, it was a natural extension of a technology the company was working on at the time. Nowadays, Docker has become an integral part of Information System Architecture and is so popular that it is almost impossible for a developer not to know anything about this topic. Docker containers utilize the Linux kernel for its functionality. Containers run in separate namespaces to provide an additional level of isolation. In the Docker environment, a client application, either the Docker engine or desktop interacts with the Docker daemon via REST API to perform operations such as building images and running containers. Thus the system architecture is a standard client-server model. The most crucial elements of Docker are images and containers, which we will describe later in detail. In addition, registries for storing and sharing images are also essential in terms of Docker. [6] The default image registry

is Docker hub, but there are numerous options out there. An image registry can even be hosted in a container on a local machine as well. [7]

Containers, as well as virtual machines (VM), are very often compared to each other since they are both virtualization tools. They still differ from each other in many respects. (See Figure 1 for architectural differences.) VMs take advantage of a hypervisor, so the hardware is being virtualized in order to run multiple OS instances, while containers provide a way to virtualize an OS. This approach allows multiple workloads to be run on a single OS instance. They both have their special areas and rather complement one another than compete. [8]

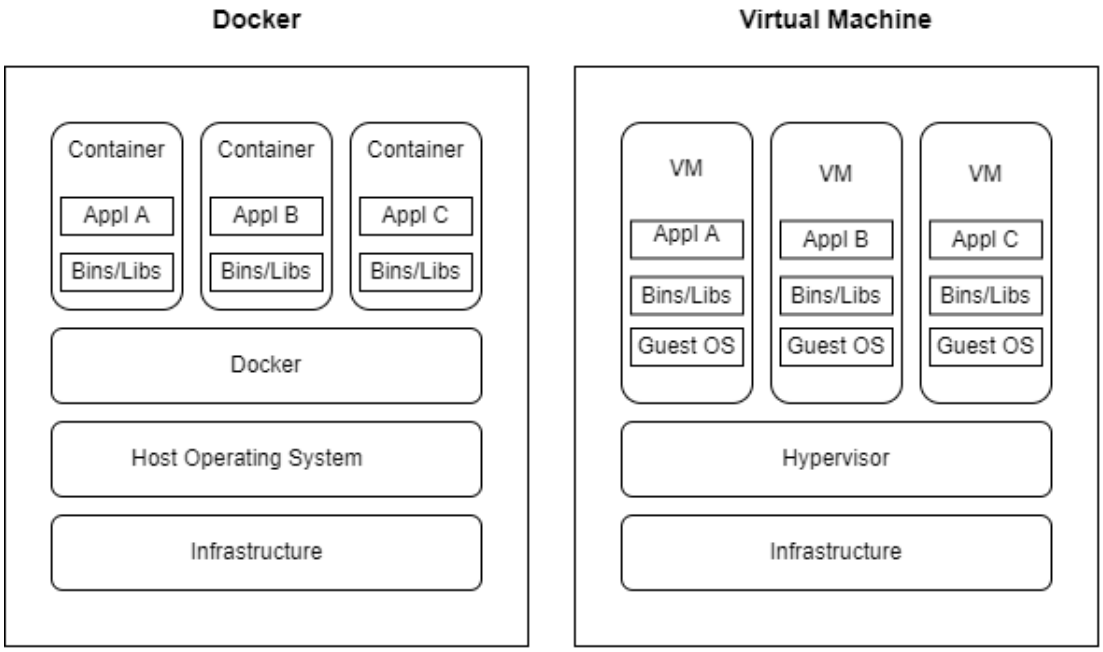

Figure 1. Docker is compared to a virtual machine. The main difference is that each virtual machine contains an operating system, while Docker runs inside an operating system.

Software development always comes with challenges. Carl Boettiger in [9] describes four of these challenges to which Docker may be the solution to alleviate the process.

The first challenge is a widely known and severe issue affecting numerous software. A study conducted in [10] shows that a significant percentage of software could not even finish the build or suffered other issues at start-up or at runtime. The configuration file for Docker images, Dockerfile, includes predefined instructions for building images. Dockerfile can be written in a way that removes this issue. However, it is noteworthy that Dockerfile can also lack information on precise build instructions like vague versioning. For example, the image tag "latest" may not be the same now as it was a year ago. Moreover, of course, it dramatically impacts how the image is finally built.

The second challenge to overcome using Docker is "imprecise documentation." With Docker, obscure documentation is not the case since the exact steps of image build are written in log files. Moreover, Dockerfile is easy to read and contains version numbers and commands to install additional tools, for instance. Further customization of Docker images is also reasonably straightforward.

The next challenge, "code-rot," is tackled with Docker's tarball feature. This feature allows images to be saved as a tarball file and read by another docker client. It is quite convenient when there are troubles with the image. Differences between the tarball file and the problematic image can be inspected using tools that Docker provides.

Even though a carefully designed DevOps pipeline may address various challenges of "barriers to adoption and re-use," the inconvenience of learning multiple irrelevant tools is an issue. Docker enables developers to use tools they find suitable rather than having them adopt unfamiliar tools and workflows. [9]

Docker containers do not persist data by nature. As soon as the container is stopped and removed, all data is lost. This default behavior is caused by a writable container layer, which is tightly coupled to the host machine. This very layer is used to store the data (depicted in figure Figure 2). Nevertheless, there are two ways to persist data; volumes and bind mounts, which require manual configuration. The most distinct difference between these two options is the location of a host machine where data is stored. By using volumes, which is the recommended way, the data is stored in Docker managed directory, while bind mount can be anywhere in the host machine. Another drawback of bind mounts is the lack of ability of docker container processes to change host file system.[11]

2.3.1. Images and Containers

In Figure 2 is an example of a Dockerfile, which is just a regular text document that includes all the instructions for building images. Very often, even the most simple Dockerfile contains instructions (FROM, RUN, WORKDIR, COPY, and ENTRYPOINT).

There are, of course, a wide variety of other instructions as well for image customization. Context and Dockerfile instructions are built as images by the Docker daemon. The context could be defined as a bunch of files at a location specified by a user. [12]

Docker image has a layered architecture, resulting from instructions in the Dockerfile. However, only instructions that make changes to the file system create a new layer. For example, the command "LABEL" changes only metadata and thus does not create a new layer. In addition, layers do not contain the same data among themselves, but each is a set of differences, so a new layer is always something new. Layered architecture

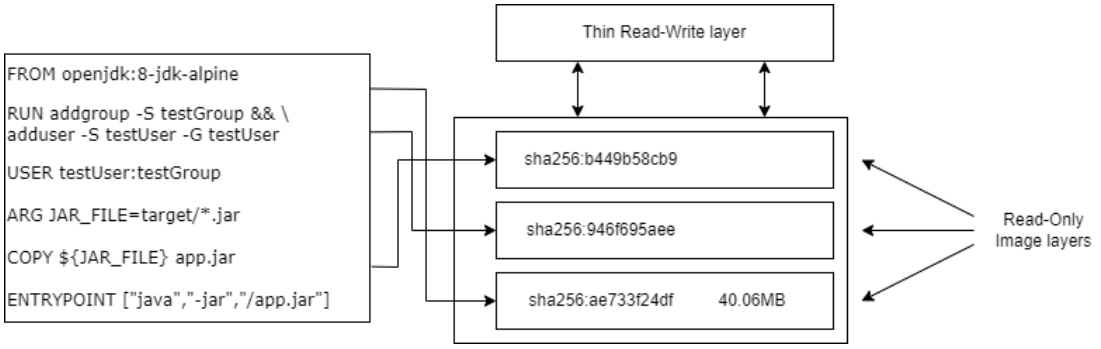

Figure 2. On the left side are presented docker commands for creating an image. Each instruction that changes the file system creates a new read-only layer. These layers are on the right side. In addition, when an image becomes a container, a writable layer is assigned to it.

decreases the workload of building, pushing, and pulling images since only layers that have changed are reprocessed.

The creation of a new container creates a new writable layer on top of the stack of only-readable layers. Changes such as file system operations to a running container are, in fact, made to the topmost layer. This "Thin Read-Write layer," depicted in Figure 2, is frequently called a "container layer." [13] There are various storage drivers to choose from. Before, the default was Advanced Multi-Layered Unification File System (AuFS). AuFS is able to merge the contents of multiple folders and provide a unified view.[8] Since AuFS is deprecated, Docker recommends overlay2 as a storage driver for most cases. [14]

Docker images are easy to share via registries, which can be defined as a storage and distribution system for Docker images. [7] We discussed tarball files earlier, which can be used to store images as well as be used in another docker environment. Nevertheless, let us focus on the registries since it is the preferred one. The Docker client includes the command "push," which is enough to share an image. The default registry of Docker is Docker Hub, but it is not limited to that.[15] Open Container Initiative (OCI) is a project for creating standards for container formats and runtimes. At the moment, there are three specifications: Runtime, Image, and Distribution specification. The work of OCI enables using multiple registries from different vendors. [16]

Docker has a tool called Docker Compose, created to ease the workload in configuring and running multiple containers. For the configuration, Docker Compose uses YAML files, where it is possible to configure volumes, networks, ports, and a bunch of other things. All the services defined in the file can be run with a single command. [17]

2.3.2. Container Networking

Docker networking is extremely complex and well-hidden from regular users. Even containers are not aware if they are running in the Docker environment. There are various network drivers available for Docker networking. Since the bridge driver is the only relevant for this project, the rest of the drivers will not be presented here.

A bridge is the default network driver; thus, if none is specified, Docker will create one. Among other things, Docker networks provide more security by adding a layer of isolation. All the containers within the same network are able to communicate with each other using the container name if the "link" configuration option is specified. However, it is optional to use the container name for communication; Internet Protocol (IP) addresses also work fine. Regardless, it is not recommended since IP addresses can change on the fly, while names remain the same as configured. On the other hand, containers in separate networks cannot see each other and hence cannot communicate by default.

There are two types of bridge networks. In addition to the automatically created default network, users can also define a bridge network. Docker highly recommends user-defined networks over the default ones. By defining a network, the "link" option becomes optional, and containers can talk to each other because of the Domain Name System resolution between containers. User-defined networks provide better isolation and control since containers need to be added manually to the network, while the default is assigned automatically if none is specified. By defining the bridge network, containers that do not need to communicate can be made invisible to each other. Other advantages include attaching and detaching the network on-the-spot, configurable bridge, and environment variables. A user can define the network manually using docker command line tool or by adding configurations in the Docker Compose YAML file. The Bridge network is depicted in Figure 3.

2.4. System Architecture

Architecture defines a blueprint for the system, and proper architecture can solve numerous issues. Also, a wide variety of challenges during the software life cycle can be encountered. Even though the life cycle varies, the process involves some common characteristics to which proper architecture provides some support. Systems often comprise multiple elements working together, providing enhanced performance, more security, and better maintainability. A basic architectural model, client-server architecture, contains at least some client application, which talks to a server-side application, which then talks to some data store. However, modern applications have a vast set of requirements, forcing

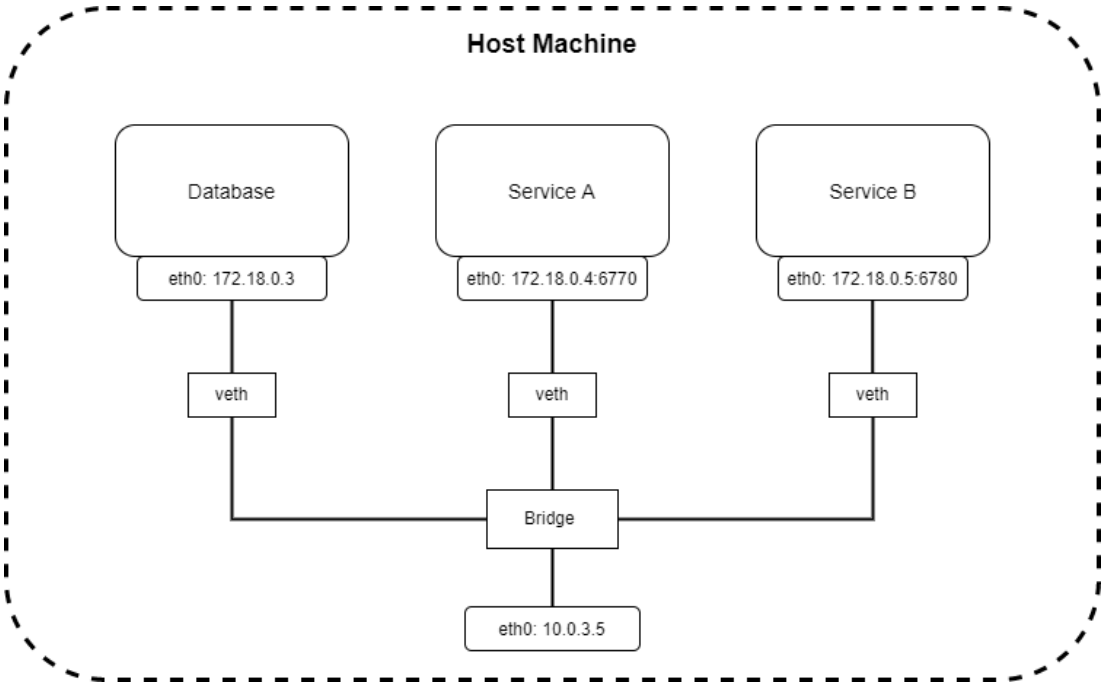

Figure 3. A network created by Docker. All services are connected via a bridge network driver and can communicate with each other. The bridge is connected to a host machine enabling connections from the host to the docker network.

developers to come up with more advanced solutions. Microservices are a way to construct systems that consist of several services, each implementing the Single Responsibility Principle (SRP). The event-driven model describes the flow of data in the system, storing mechanisms, and actions triggered. For example, instead of keeping only the latest value of an item in a data store by overwriting it, all modifications create a new event implying what happened.

Microservices and event-driven models are described in detail in the following sections.

2.4.1. Microservices

A distributed system loosely defines autonomous computer systems linked via a network. Each computer runs software enabling communication and data sharing between these computers, often referred to as nodes.[18] Hence, microservices, as well as other architectural design patterns consisting of smaller components that finally create a fully functioning entity are a subset of distributed systems.

Microservices apply SRP, and each service’s boundaries are distinct, but they are loosely coupled among themselves leading to a very beneficial outcome. Services work independently; further, modifications

to one service will not affect another. This principle also promotes independent scaling and deployments. [19] Since an official definition of microservices does not exist so far, possible approaches or best practices for building microservice-based systems are described by Independent System Architecture [20] according to the following nine principles:

1. All modules must provide an interface for interaction between other modules.
2. Two different architectural models, from which the first one covers decision that applies to all modules. And the second one covers decisions for each of these modules independently.
3. Modules are always independent and separated processes, containers, or VMs.
4. Standardized service-to-service communication.
5. Standardized authentication.
6. Continuous delivery pipeline for each module.
7. Standardized operations, for example, logging and alerting.
8. The standards mentioned above should apply on the interface level, but the REST implementation inside a module is specific to that module.
9. Modules must be able to compensate for other modules in case something unexpected happens.

Many Information Systems are moving away from monolithic architecture towards microservices. It is not undeniably better, but it for sure works better in some cases. Members in the study group in [21] did or did not vote for these subjects pointed out. Advantages and disadvantages are not equally weighted. Thus some are voted as a more significant advantage or disadvantage. See the list of advantages and disadvantages over monoliths below:

- **Advantages**
  - Ability to scale up or down with ease.
  - Distinct boundaries in responsibilities.
  - Independently deployable services.
  - Rapid feedback loops.
- **Disadvantages**
  - Requires experienced developers.
  - Increased level of difficulty.
  - Complexity.

2.4.2. Event-Driven System

While microservices describe service-oriented architectures, event-driven system (EDS) describes the data flow in the system and how it is generated. Thus microservices are able to utilize this concept of EDS.

The working principle of an EDS differs from traditional request-response systems because it indeed operates with event-based data, indicating what happened rather than pointing out the current state. In contrast, stereotypical traditional systems respond to direct user requests, and updates to data override the existing data. EDSs include three loosely coupled components: producers, consumers, and data processors. A Data processor takes input from a producer, processes the data, and produces the data in a form that might be useful for some consumers in the system.[22] Also, Brenda M. Michelson in [23] brings out one more layer, an event channel, which acts as the backbone of the event-based systems by providing a route for events between producers and consumers.

Events consist of two parts, header, and body, where the header part contains general information such as an id, type, name, or timestamp of an event. In comparison, the body describes why the event occurred.

Multiple services form event-driven architecture, and event producers are unaware of other services. Hence they generate an event and do not care what happens to the event after that. There are three types of event processing: simple, stream, and complex. Complex processing contains characters of both; simple and stream processing, including more advanced event handling to respond to emerged threats or opportunities, for instance.[23] This project utilizes EDS to create a data stream from the PDM system, and finally, deliver it to users after via a data processing service.

2.5. Data Stores

Data can be stored in various ways. However, the continuously increasing amount of data has created a need for dedicated data stores specialized for storing, ordering, and efficiently querying data. Furthermore, data stores are running continuously, hence providing high availability. Some emphasize speed over other features, and some might be specialized for a massive amount of data. Nonetheless, even if the data storage requirements are precise, multiple data stores assumably meet those requirements. This chapter overviews NoSQL databases, including persistent and in-memory data stores.

Databases have mechanisms for streaming changes as they happen or triggering actions based on changes in a database. Although, this project takes a different approach, which will be discussed later on.

2.5.1. NoSQL Databases

As mentioned earlier, data has become such an important factor in the business area that already, in 2016, there were more than 225 NoSQL databases to choose from. NoSQL stands for "not only Structured Query Language (SQL)"; hence covers all databases that do not use SQL as a querying language. [24]

Document-based databases, sometimes called document stores, use JavaScript Object Notation (JSON)-like documents to store data. This document is equivalent to a table used in relational databases.[25] One subcategory of NoSQL databases is Document stores, and they have various advantages like horizontal scalability enabled by distributed architecture, schemaless data structure, and user-friendly way to interact with the database. [26]

Also, one of the most popular document stores, MongoDB, was put to a test in[25] against Oracle database in terms of speed of database operations, resulting in MongoDB winning overwhelmingly. [25] states that in a case where relations between tables are needed, the Oracle database can be relied on. However, MongoDB supports relations as well [27].

NoSQL databases are sometimes used as in-memory data stores. By storing all data in Random Access Memory, the speed of operations increases significantly. Although, the downside of in-memory data stores is volatility [24]. In case of service crashes, all data is lost unless it is not explicitly persisted.

One notable in-memory database is RocksDB, created by a team at Facebook on top of LevelDB. RocksDB is built for embedded workloads, but nothing prevents us from using it as a regular data store. [28] Kafka Streams, which is a significant part of this project, uses RocksDB by default as a key-value store [29].

2.6. Kafka Ecosystem

Apache Kafka (called Kafka from now on) is an open-source event streaming platform capable of handling trillions of messages a day due to its ability to scale horizontally and optimization for a high message throughput. Originally Kafka was created at LinkedIn as an in-house project to move data around. However, LinkedIn open-sourced it on GitHub at the end of 2010. Since then, a lot has happened to that project. It carries the name Apache Kafka, and some of the largest companies in the world have adopted Kafka as their event streaming platform. [30]

Kafka is really the only player in that area, even though multiple message streaming platforms, data processing software, or data integration tools are constantly compared to it. The fact is that Kafka is able to perform all these tasks. It acts as an integration point between applications without

coupling these applications tightly, stores data as long as required, and offers a native data processing tool, Kafka Streams API.[30] This all might sound perfect, however, Kai Waehner lists in his blog some cases in which Kafka is not best suited. For example, "hard" real-time events (as opposed to soft real-time events) like car engine control that require constant real-time data flow. Despite the power of Kafka, there is a delay in data transfer between producer and consumer, which cannot be tolerated.[31]

The Kafka ecosystem is highly configurable and consists of several services or components, some of which are optional but relative to this project. These services and components will be presented in this chapter.

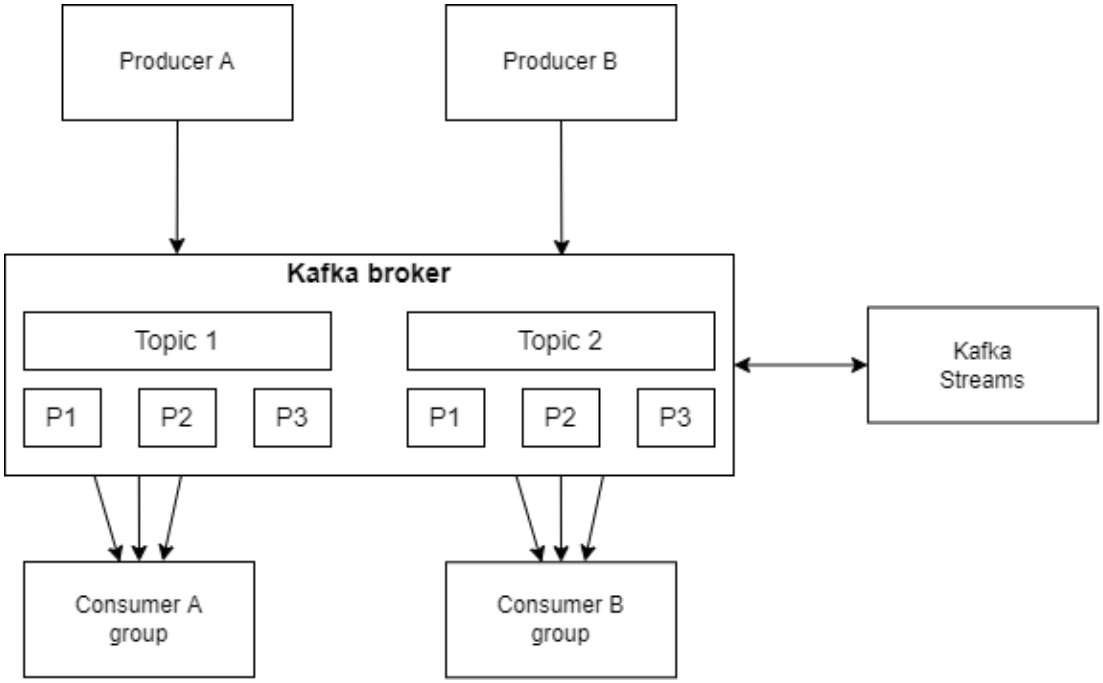

Figure 4. Simplified version of the Kafka producer-consumer model. Producer produce messages to Kafka topic, which is then consumed by consumers. Topics consist of one or several partitions (P1, P2, and P3 in this case), which are allows distributing logs between multiple broker instances.

2.6.1. Kafka Broker

The main component of Kafka is a single Kafka broker (see Figure 4), but the system can be scaled out by running a cluster or even several of them containing multiple instances of brokers. Brokers in a cluster are able to communicate with each other and thus elect a so-called leader broker to work as a cluster controller with responsibilities like partition assignment and failure detection. Albeit brokers are designed to work in clusters, with proper configuration, even one broker can handle vast amounts of

messages. Although, single broker configurations lack helpful features like data isolation in terms of security.

Broker is the part in Kafka that takes messages as input, assigns offsets, and then writes messages to storage. The broker also sends the data to consumers when requested.

One of the core features of Kafka is retention, meaning that data on a topic can be stored as long as required or by the log size. As soon as configured retention time has passed or the log has reached its maximum size, messages will be deleted respectively.

2.6.2. Producer-Consumer Model

We described the data flow of the Kafka ecosystem already in chapter 2.4.2. However, the implementation details are Kafka-specific. Thus chapter 2.4.2 does not include them.

Messages are routed using topics. The producer produces a message to a topic, and a consumer reads the message from that topic. To read the right message, the consumer needs to keep track of the current offset, which is basically metadata added by Kafka for tracking the specific location in queue where to read. By default, the consumer reads from the end of the message queue, but the desired reading point, offset, can also be specified. The offset will come in handy when a consumer crashes since the offset can be retrieved from the Zookeeper or broker after a restart. The purpose for Zookeeper is explained in section 2.6.5. Often there are several consumers, and each consumer reading messages from the same topic belongs to the same consumer group. This group takes care of all partitions of the topic by sharing responsibilities. Partitions are storage units or message logs of which topics consist of. See figure 4 For example, if a topic has two partitions, and three consumers read from that topic, then two consumers read from one partition, and the third one from the second partition. *Partition* is a data log containing a topic's messages. One topic is often divided into multiple partitions to increase performance.

2.6.3. Kafka Connect

Kafka Connect is a specific consumer that can be configured using connectors. It is created for situations when data goes to Kafka from some system that cannot be modified or another way around. A great example of this is streaming changes from a database to Kafka. There are various of pre-made connectors that can be used to either move data from or to a database. For starters, to put Kafka connect in use, connector plugin files need to be installed on Kafka Connect, and then the connector itself will be created using the REST API provided by Kafka Connect. Furthermore,

Kafka Connect has the ability to convert data from one format to another using Kafka or Confluent Schema Registry. Schema Registry is a separate service from Kafka and is built to store schemas and their history, thus enabling schema evolution as well. Some data formats work with Kafka out of the box, but some of them, like Avro or Protobuf, require the functionality that Schema Registry provides.[32]. Schema Registry does not come with Kafka, but multiple open-source options are available. [30] Figure 13 in appendix 1 shows how Schema Registry is placed in the system architecture. An example of a JSON formatted message sent by the Debezium connector presented in listing 2.1.

2.6.4. Kafka Streams

Kafka Streams is a specific consumer as well, which is also able to act as a producer providing a native way of processing data streams in Kafka. There are two APIs to choose from: low-level Processor API or high-level Domain-Specific Language (DSL). DSL includes functionality for filtering, grouping, joining, windowing, and several other processing needs. Hence it is an excellent option for many Stream applications. Kafka streams applications are both producers and consumers. They fetch messages from a topic just like any other consumer and also produce processed data to topics.

2.6.5. Zookeeper

The last component in the Kafka ecosystem is called the Zookeeper. It provides a way to keep track of metadata of topics and partitions [30]. However, Zookeeper comes with drawbacks, such as increasing complexity and duplication. For these reasons, the preferable way to store metadata in the future is using Kafka. Furthermore, at some point, Zookeeper will no longer be supported [33]. Figure 13 in appendix 1 shows how Zookeeper is placed in the system architecture.

Listing 2.1. An example of a JSON formatted message sent by the Debezium connector. This is an update messages to an existing document, which includes values before and after update as well as some metadata about the source.


```
{
    "schema": { ... },
    "payload": {
        "before": {
            "event": "status change",
            "code": "ABCD987",
            "type": "document",
            "version": "2.0.1",
            "author": "TTV123",
            "description": "",
            "status": "In process",
            "date": "03.12.2022, 13:30:15"
        },
        "after": {
            "event": "status change",
            "code": "ABCD987",
            "type": "document",
            "version": "2.0.1",
            "author": "TTV123",
            "description": "",
            "status": "Rejected",
            "date": "24.12.2022, 08:00:23"
        },
        "source": {
            "version": "2.0.1.Final",
            "name": "server1",
            "ts_ms": 1520085947253,
            "txId": "6.9.809",
            "scn": "2125544",
            "commit_scn": "2125544",
            "rs_id": "001234.00012345.0124",
            "ssn": 1,
            "redo_thread": 1,
            "user_name": "user",
            "snapshot": false
        },
        "op": "u",
        "ts_ms": 1532592859002
    }
}
```

3. REQUIREMENTS AND DESIGN

In this chapter, walk through the system design and give an idea of the project. We will discuss the requirements thoroughly, followed by architecture and technologies. This project is about building a system, let us call it "the Watcher," capable of streaming changes from the PDM system and delivering them via email or as a push notification to a client application. The changes to stream to users are based on filters that users have set by creating subscriptions.

3.1. Background

To put this simply, a system built over two decades ago has reached the point where it does not meet the requirements for software today. The requirements related to this project are presented later in the chapter Technical Requirements 3.2.2. Hence, this legacy system must be rebuilt using state-of-the-art technologies and architectural patterns.

There are various general reasons for replacing legacy systems (LS). A study conducted in [34] brings out several great standpoints on this matter. The most significant drivers for starting the modernization process include inflexibility, lack of expertise and documentation, as well as high maintenance costs of LS. Furthermore, it was stated that software should be brought to market faster than is usually possible due to the inflexibility of LS.

3.1.1. Legacy Subscription Service

Originally, the old system, let us call it as "the Legacy Subscription Service" (LSS), which will be modernized during this project, was built to provide information automatically about changes in the PDM system by working as an engine in the background. Many LSS functionalities are currently embedded into the PDM system, even though LSS could be a separate application rather than a necessary part of the core system. The LSS is mainly built on database triggers, procedures, and functions. However, it also contains a server-side application that runs scheduled tasks.

In addition to the various reasons listed above, the new version should be easy to plug in due to customers' different business requirements.

The goal is not to build a copy of the previous system with newer technologies. The goal is to meet the functional requirements set to this system (described later in the next section) and to keep our business needs in mind today. Thus, rather than modifying the old code, the new system will be built from scratch, including the architecture.

The study in [34] points out some challenges that should be prepared for, such as time requirements and data migration.

3.2. Project Requirements

The Watcher will find its place in our selection that comes with the PDM system. Therefore, there are requirements needed to be taken into account when designing the system. While the main goal is to build software for customers that helps their business run and listen to them in case of any special needs, the base of the solution ought to fit multiple different purposes that may emerge later on.

However, despite the potential for further development, we build this system to monitor the PDM system for changes and notify users by sending emails or push notifications according to subscription details. A subscription contains information on how users want to get informed, comprising filters such as document code, status change, and what operations are done to the document. For example, a user may want to receive a notification immediately when the status of a document changes to "accepted" but does not find the other steps; status "rejected," for example, of document life cycle important.

In the following two sections, we will first describe the functional and technical requirements for the new system.

3.2.1. Functional Requirements (FR)

The original idea of LSS was to have a service that notifies users about changes in the PDM system they are interested in. The list of features has grown over the years based on customer feedback and the findings of our research team. All the features of LSS have proven useful. Thus most of the functional requirements for this project come from the LSS. Our team keeps searching for new features as well as customers are encouraged to ask for further tailored development. Initial functional requirements for the project are listed as follows:

1. **(FR1) Informing:** Receiving document updates via email or as a push notification.
2. **(FR2) Edit subscriptions:** Subscribing or unsubscribing for a subscription.
3. **(FR3) Create a subscription:** Creating new public and private subscriptions.
4. **(FR4) Browse subscriptions:** Browsing available public subscriptions as well as own private subscriptions.

Listing 3.1. create subscription

```
{
    "subscriptionName": "Updates for the progress of
       subscription service"
    "objectType": "DOCUMENT",
    "document": null,
    "documentGroup": "subscriptionService",
    "version": null,
    "subscriptionCreator": "TTV123",
    "restrictions": ["NO_RESTRICTIONS"],
    "triggeringEvents": ["DELETED", "STATUS_CHANGED"],
    "statuses": ["TECHNICAL_ERRORS", "REJECTED", "
       READY_FOR_PRODUCTION"],
    "deliveryType": "EMAIL",
    "publicVisibility": true,
    "combined": true,
    "cron": "0 8 * * 1,3,5"
}
```

5. **(FR5) Create filters:** Add filters on the subscription, which will determine received updates. (See the JSON document 3.1)

6. **(FR6) Admin features:** Admin user is able to add other users to a subscription.

7. **(FR7) Emails:** Information is added dynamically on email templates, and they should look appealing. Alternatively, the information is available on a specific endpoint.

8. **(FR8) Data migration:** The new system has to be able to use subscriptions created for the previous system.

Subscriptions (FR2, FR3) are a set of rules, and possible options for the rules are managed by companies. However, options applied to a subscription will be defined by a subscription creator. A rule could be a time interval for receiving emails, for instance, or the form of notification. The interval will be set using cron expression and the form by choosing email or push notifications. Subscription message is presented in listing 3.1.

3.2.2. Technical Requirements (TR)

In addition to the list of functional requirements, which explain what users can do with this application, technical requirements are imperative in

describing how things are done and therefore need a closer inspection. The design process started with marking the system's needs as a whole. Even though there are strict guidelines for functionality, the technical entirety of the project is the result of creativity and a vast amount of background work.

First of all, software tend to get old at some point, which is what happened here as well and why this project exists. These old systems often come with multiple downsides, like poor scalability and performance issues [35]. No matter how the system is built, it will likely become obsolete in the future. However, one goal for this system is to build it in such a way that it will remain robust. Related technical requirements comprises **proper feedback (TR1)**, **reliability (TR2)**, **fault tolerance (TR3)**, **safety (TR4)**, **scalability (TR5)**, and **maintainability (TR6)**.

(TR6) includes, among other things, clear architecture as well as clear documentation. All developers should be able to grasp the functionality quickly and understand how the system works on a deeper level.

The system will be built from several more minor services to overcome common challenges, such as scalability. **Single-responsibility principle (TR7)** will be taken into account in the design and carried out as well as possible without adding excessive complexity. Microservices scale out well, which is great since it allows adding more power where needed. In contrast, monoliths can scale only as a whole. This sort of loose coupling is desirable for several other reasons as well. For example, even if one service goes down for some reason, other services are most likely capable of performing their processes. On the contrary, tightly coupled services are often heavily dependent on each other, leading to a situation where the whole, or at least a more significant part of the system is down.

Modern web applications should work on **various platforms (TR8)**, even though it would be optimal to build it for one thoroughly tested platform. However, some customers prefer an on-premise solution, while others favor a cloud-based solution. Operating systems can differ as well. Nonetheless, there should not be any trade-offs regarding reliability; the system should work seamlessly in both situations. Containerization has proven its benefits in various environments, and it seems to be the best solution for this requirement.

Persistence (TR9) for some of the data is crucial, which boils down to one thing, databases. Even though we will allow some data to be stored in volatile memory for quick access, eventually, these types of data stores, such as cache, rely on persistent databases to ensure integrity.

This system does need to provide functionality for authentication. However, all the endpoints are protected, requiring authorization. Therefore, we will operate with an existing authentication service to handle that part.

3.3. System Architecture

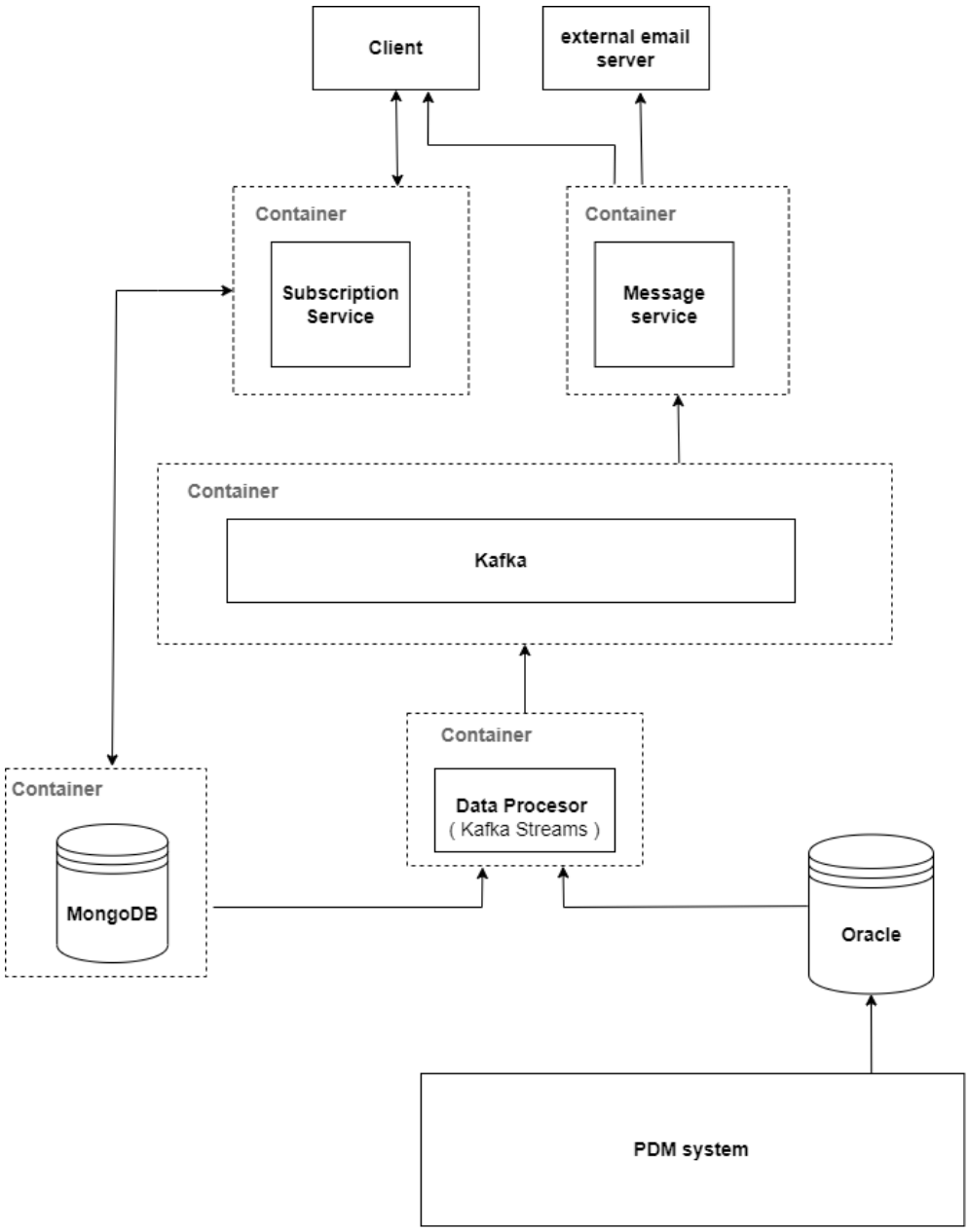

Figure 5. Simplified system architecture. Data from the main system is stored in the Oracle database (lower right). See appendix for full architecture

Initially, the idea was to have a REST API in one service that accepts user requests and another responsible for publishing event changes to comply with SRP. Even though this pattern does not quite check all the boxes of Command Query Responsibility Segregation (CQRS), the design is influenced by it. This pattern, CQRS, was coined by Greg Young [36]. CQRS is built on top of the Query Separation Principle, defined by Bertrand

Meyer. Meyer states that if a query returns data, it cannot mutate data. [37]

This project will take advantage of both concepts. Therefore the service containing REST API has endpoints for creating and updating data, as well as querying data. None of the actions mutating state return data. On the other hand, queries that return data do not mutate it. CQRS pattern comes into play when original data created through the REST API gets modified before users are able to fetch the data. Therefore, the write-object model differs from the read-object model.

CQRS pattern works well with EDSs. [36] Even though it all starts with a simple API post or put request, it will end up in the EDS that finally sends or pushes out the processed data. The original request could be described as a filter object since data that gets out of the system is defined by that filter object. The architecture that CQRS influences is presented in figure 6.

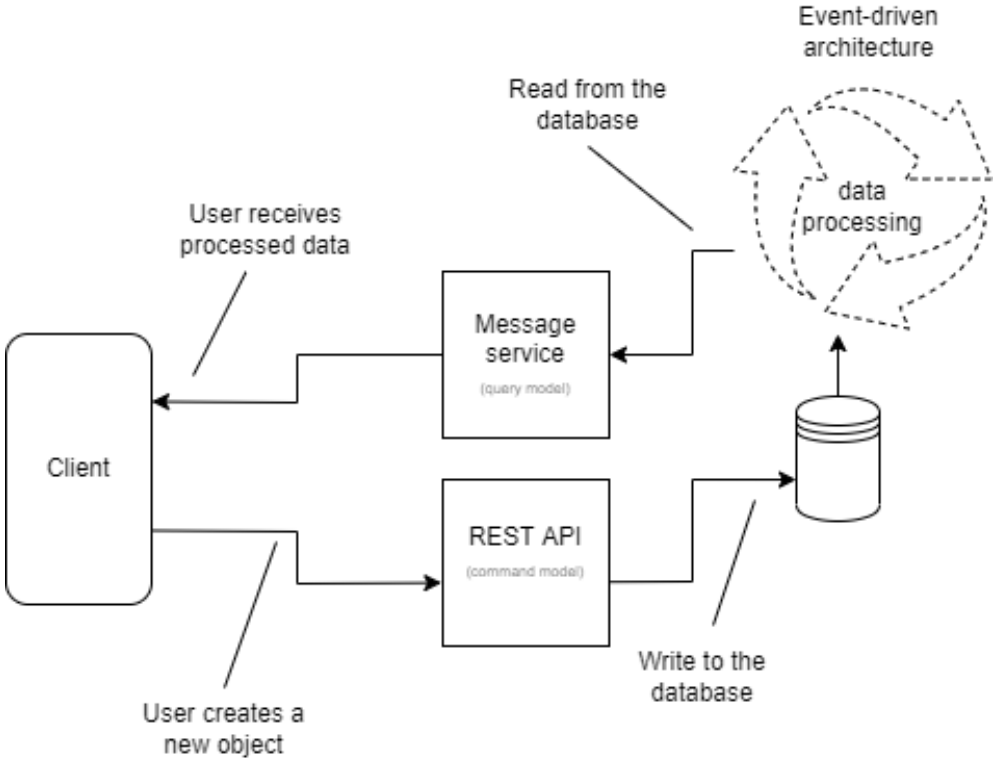

Figure 6. Diagram represents the part of the system that utilizes the CQRS pattern. Writing to the system is done through a REST API and reading through a message service.

Although the client application is too much work to fit into this thesis, it is good to mention that there will be one later. More precisely, the client application will be built and integrated as a part of an existing application. Also, we do not need to provide functionality for searching objects since the

application will get information such as document codes or filtering options from another service.

3.3.1. REST API

We will use Spring Boot, the most popular Java framework to build the REST API [38]. In addition to its popularity, Spring Boot lines up well with the technology stack we use in other projects; hence was chosen for the project. The REST API allows users to create brand-new subscriptions by specifying what sort of information they would like to receive. Listing 3.1 presents an example of a new subscription.

Subscriptions cannot be edited after they have been created due to how the system has been structured. For example, suppose a user has subscribed to get information on a specific document group with a particular rule set. In that case, it could be extremely harmful if other subscribers were able to change the rule set. However, the list of subscribers in the subscription model can be modified, which allows users to subscribe or unsubscribe from an existing subscription. Deleting a subscription could also be harmful; thus, the current design will not let it. Other usable endpoints include, among other things, querying existing subscriptions, checking the health status of the service, and verifying the running software version. Example of endpoints presented in tables 1 and 2

All endpoints must be protected, and a user's identity must be verified before using the API. The authorization token will be forwarded to an external authentication service in every API call, which then responds with user status. Since there are more features for admin, auth service has to separate normal users from admin users.

GET	/api/v1/subscriptions/{id}?pageNum={num}
PUT	/api/v1/subscriptions/{id}
POST	/api/v1/subscriptions

Table 1. Endpoints for retrieving, updating, and creating subscriptions.

GET	/api/v1/service/status
GET	/api/v1/service/info

Table 2. Endpoints for checking service health status and current service information.

3.3.2. MongoDB

We chose MongoDB[3] as the database for subscriptions since the data structure of subscriptions transforms well into documents. Subscription data will be divided into four collections in total due to the slightly varying content of different types of subscriptions. Two MongoDB collections will contain subscriptions subscribed by a code of a single object such as printer documentation, and the other two by a code of object groups such as sales documents. Then these are again divided based on the delivery type; email or push notification. Although, it would be possible to store these four slightly different groups of subscriptions into one collection, this approach simplifies data processing at later stage. On the other hand, it is a trade-off since we need to create four distinct models and handle them separately.

3.3.3. Kafka

First, Kafka[4] was not an obvious choice. While Kafka can handle massive amounts of data, it also requires more research to be implemented successfully. RabbitMQ is a widely used message broker and much simpler overall. However, we were soon convinced by Kafka's functionality, such as native stream processing support and premade connectors for streaming changes from databases into Kafka topics. Also, Kafka has a large online community, which is always a huge advantage. Lastly, our customers are companies of various sizes; thus, the system needs to be able to scale, and Kafka is basically built to do so.

In the system, each service has its own specific and essential task. However, the Kafka broker, or several depending on the configuration, is the center of the system. All data will be published into Kafka topics and consumed from there by another service. There are a couple of essential things to pay attention to; the Kafka broker does not do all the work alone. Kafka Connect[5] service and connectors applied to it, Zookeeper, and Schema Registry play a vital role in the Kafka ecosystem.

Neither a Zookeeper nor Schema registry is necessary. However, messages in Kafka broker are in binary format. Thus they need to be serialized or deserialized for other services. There are multiple data formats to choose from, like Avro[6], which is also used throughout the project. One of the most significant advantages of Avro is schema evolution, meaning that schema can change in the future without becoming obsolete. Because it is impossible to say if the schema will change, this is the safest pick and, thus, part of future-proofing the application. Avro is not supported

[3] https://www.mongodb.com
[4] https://hub.docker.com/r/confluentinc/cp-kafka/
[5] https://hub.docker.com/r/confluentinc/cp-kafka-connect
[6] https://www.confluent.io/blog/avro-kafka-data/

out of the box; hence Schema Registry is required for this setup. Also, leaving out Zookeeper is not recommended yet in production when the latest Apache Kafka version is 3.3.

Kafka Connect is running on a separate service and stands in front of the Kafka broker itself. It is highly configurable via a REST API by applying connectors to it. There are numerous connectors out there, so developers do not need to invest in them. The system needs to stream data from two different databases. Therefore two different connectors are required.

There are a few ways to watch database changes and get information about those changes. This project will utilize a free-of-charge Debezium Oracle source connector, which captures and records row-level changes using Oracle's native LogMiner database package. [39]

The first option was MongoDB's official connector for streaming data from MongoDB. However, it does not support snapshots, which is a must-have feature. Without it, subscriptions created before starting the Kafka broker would not be transferred to the Kafka broker. In addition to the Debezium Oracle connector, Debezium has a connector for MongoDB, which supports snapshots. The documentation is quite comprehensive for both connectors, including clear instructions for specific configurations.

3.3.4. Data Processor

Simple data processing could be done by adding configurations on Kafka broker. Although, that does not offer enough functionality in this case. Another way of processing data is Kafka Streams API, which comes with high-level DSL and low-level Processor API. This project benefits more from simple DSL than getting down all the way using Processor API.

Using Kafka streams does not require anything other than just using its functionality; thus, data processing could be located in a separate service specialized in data processing as well as in some existing services. To scale up easily later on, we find it convenient to use the first option and create data processing functionality in a separate service. Kafka Streams data processor service is most likely the service with the most considerable amount of traffic, which also supports this decision. This structuring facilitates debugging and maintenance while all the data processing logic is placed in one service, which consumes data from Kafka topics, processes it, and produces it to a new topic.

The Data Processor service is stateful since it will utilize key-value stores that come with Kafka Streams. Moreover, we will store all users and subscriptions in the Streams application's default key-value store, RocksDB. There are two reasons to do so; faster access to data and pre-processed data. The Data Processor will be under a heavy load, which would even add more burden if subscriptions were queried every time a new event occurs.

In list 3.3.4 is explained how the application starts and the desired initial state is achieved.

1. Data processor starts
2. Kafka user event listener starts.
3. User store is initialized.
4. User store is populated.
5. Kafka subscription listener starts.
6. Subscription store is initialized
7. Subscription store is populated.
    - Before subscriptions are stored, they will be enriched with a piece of user information such as email addresses.
8. Kafka event change listener starts.

After all eight steps are done, a stream of events starts coming in and will continue until the service stops.

3.3.5. Message Service

The Message Service has one primary task: send data to users. However, the data can be sent via scheduled emails or as push notifications. The Message Service is also a Kafka consumer; thus, data is fetched from Kafka broker topics.

There are two push notification options: WebSockets or Server-Sent Events (SSE). At a minimum, it requires a connection with a client application, and data is only forwarded from Kafka to the client application via the Message service. The initial plan was to use WebSockets, but there is no need for two-way communication or other great features that WebSockets provide. Hence, the better fit is SSE. This pipeline will immediately stream events from Kafka topics to clients without storing them.

Scheduled emails instead are somewhat more complicated. Fortunately, there is an impressive library for scheduling events, Quartz. For Quartz, the data store has to be specified. Options to consider were MySQL, H2, and ramJobStore. However, ramJobStore does not persist data, so it is excluded. MySQL persists the data, but it would run in a separate container, which is not desired since containers are starting to pile up. This leaves us with the last option, H2, which can be configured to persist data and run in embedded mode.

Spring boot provides a simple way to send emails using the JavaMail library. Furthermore, there are tons of great examples of implementing the feature, even with scheduled events, which is a huge benefit. Kafka consumers will keep fetching messages continuously and storing them in the same H2 database as Quartz stores all the data. As soon as the Quartz scheduler is triggered, events will be fetched from the database, and emails will be sent to users using the proper email template.

3.4. Conclusion

This architecture has characteristics of a couple of different approaches in system architecture design. The design takes advantage of event-driven architecture since the exciting part is purely an unbound stream of events. It is also a distributed system by the nature of event-driven architecture. Implementing CQRS in full would require, among other things, the subscription service to be split into two services and provide reads and writes separately from different services. However, this approach would not have been justified due to the small size and low amount of data traffic in the subscription service. Instead, processed data will be available in the message service; thus, the architecture partially separates reading and writing.

The idea behind the design was to create a set of independently deployable and loosely coupled services that are easy to scale and maintain. Another important factor to pay attention to is how the system as a whole is integrated with the PDM system without touching the code base of that system. Also, this system can be just plugged in or out depending on the business needs.

4. IMPLEMENTATION

This chapter presents tools for the project and states why these tools were chosen as well as every step of building the Watcher. The following sections are ordered based on data flow. We are starting with the Subscription REST API, which handles all user requests to the system (see Figure 5). Kafka services are part of the data processing and watching database changes. Finally, we get to the Message Service, which informs users by sending emails or push notifications.

The system architecture 5 is quite complex compared to the basic model, where a client makes an HTTP request to the server, and server-side code may perform some data processing. Then the data is stored in the database, and the client receives a response about how it went. Since we had no prior experience with any of these technologies except MongoDB, we approached this challenge by creating a so-called test environment where all the services are connected and able to produce and consume Kafka topics. This environment does not include, for example, data processing. The data goes through the Kafka streams service and other services to verify the configurations are correct and that serialization and deserialization work properly. In addition, all data models are in a simplified form. It all started with moving strings and integers around the system and then moving forward using the Avro data format.

This approach was great since we had to adjust the architecture multiple times. We considered having only one data store but soon realized that we needed to process data as quickly as possible and use local data stores in the Data Processor to achieve better performance. Also, processed event changes should be located in the Message service since that is where we use them. These changes greatly impacted the overall architecture requiring modification throughout the system.

We also considered if the Kafka broker messages should be available for client applications directly. Still, we discarded the idea to keep the Docker network isolated from other applications and staying consistent. Later during the project, we will implement a separate service for messaging, which will be the only service for producing event change information for client applications.

One of the principles in the current design is to perform necessary operations on the data when possible, move it where it is used, and store it in its processed form. In addition, all services deal with the Kafka broker only. On the contrary, in the initial plan, all data was stored in MongoDB and retrieved directly from there instead of Kafka broker.

4.1. Tools and Development Environment

By far, the most typical Integrated development environment for developers using Java Virtual Machine is Intellij IDEA with 71.6 percent share [38]. However, we prefer lightweight and lightning-fast Visual Studio Code (VS Code) since it includes all features we may find beneficial. In addition, a massive collection of VS Code plugins is available in case something is missing.

We chose Maven as the build tool, also supported by VS Code. In addition to the building capabilities, Maven is able to do much more. Official Apache Maven Project websites [40] refers to it as a "software management and comprehension tool." Maven is not the only tool for the job. However, it is holding first place currently. In fact, it has been the most favored build tool for Java Virtual Machine -based languages for years. [38][41]

During development, we need to run multiple services from the command line. A new command line window for each service is an option. However, terminal multiplexers like Tmux allow a terminal window to be split into numerous smaller windows and further organize them as desired.

The final version will encompass several services running inside containers. Hence, it makes sense to include containerization, Docker specifically, in the project already in the development phase. Using Docker containers also simplifies the process of running multiple services since it brings Docker Compose available.

4.2. Subscription Service

This chapter details the REST API implementation, data storing in MongoDB, and the phases between them. We also go through the REST API documenting tool, Swagger[7], as well as exception handling and additional libraries. This service is required for users to be able to interact with the system by fetching, creating, and modifying subscriptions.

Table 3 shows the main technologies for the Subscription Service.

Spring Boot	Framework for the Subscription Service.
Spring web	Developing the REST API.
MongoDB	Database for persisting subscriptions.
Swagger	REST API documentation.
Lombok	Reducing boiler plate code.
Docker	Service containerization.
Maven	Building and managing the project.

Table 3. The main technologies of the Subscription Service.

[7] https://swagger.io/

4.2.1. Spring Boot

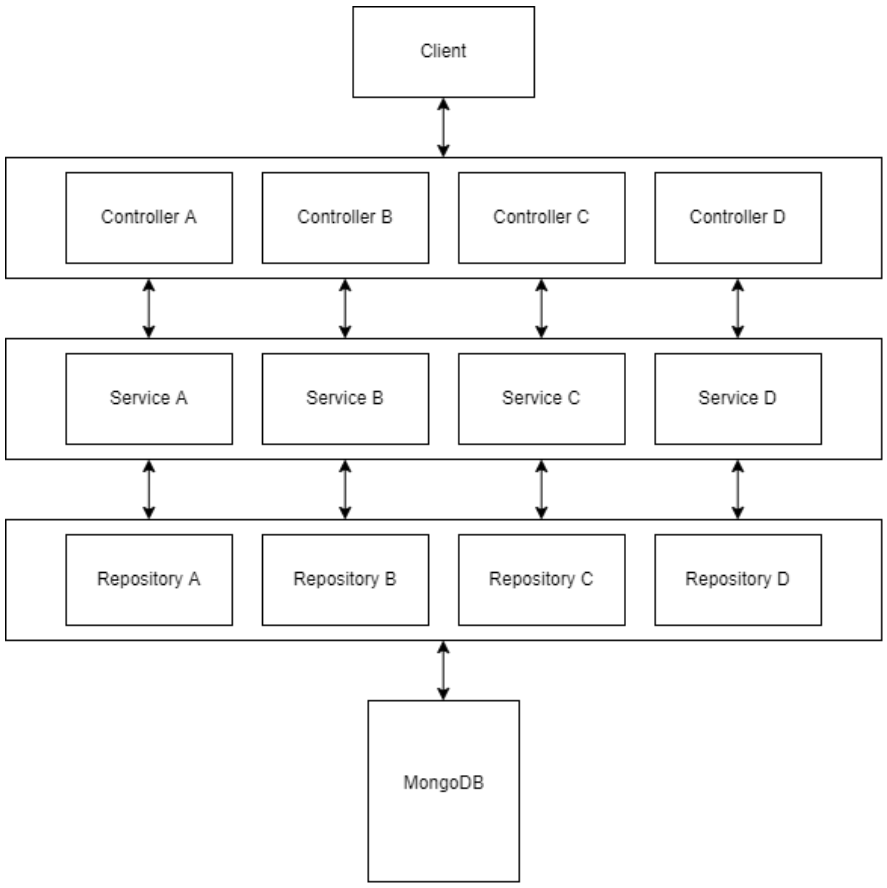

Figure 7. The layers of Spring boot application. The controllers contain endpoints, while all the logic is handled in the services. The repository is an abstraction between services and the database.

Files are located based on the layer to provide a clean and maintainable structure. For example, we place all controllers in the controller's folder, and all services are in the services folder. However, an alternative file structure, structure by feature, is presented in Spring Boot's[8] official documentation [42]. Since with Spring Boot, we use JVM -based languages, and it is helpful to follow general Java conventions as well [43]. We also need to consider REST principles 2.2. Spring Boot is a framework for building backend applications; hence "client-server constraint" is fulfilled by default. The same applies to the "layered system" constraint since the client application communicates with Spring Boot, which further communicates with the database. The same pattern can be notified throughout the whole system, thus does not limit to the Subscription Service only.

[8]https://spring.io/projects/spring-boot

4.2.2. REST API

Initially, the design included just one subscription model, which would fit all different types of subscriptions. However, by looking further, it is a better option to categorize them based on the notification delivery type and criterion. There are two delivery types available; email and push notification. The email model includes properties such as delivery interval and information about the email format. By contrast, the criterion indicates how users subscribe to the objects. There are two criteria; subscription by object codes or object groups. Eventually, we divided one subscription model into four different categories. To follow REST principles 2.2, each of the subscription models creates, modifies, and fetches subscriptions, thus have own URIs for each resource.

Spring Boot comes with useful libraries to build applications more efficiently. One of those libraries is called "web," which defines all endpoints in this API. Files that contain endpoints belong to the controller layer and are called controllers. To maintain a clear file structure, the controllers of this project are placed in the controller folder and do not contain any business logic. The business logic is in the service layer. Further, controllers call interfaces of services, not services directly, to achieve more abstraction. Services are responsible for validating input, creating objects, and calling the repository layer for persisting or fetching data from the database.

All endpoints are protected, meaning that authentication is required. Since the service does not implement any authorization mechanisms, a user needs to acquire a token from an external authorization service, and the validity of that token will be checked on every endpoint.

4.2.3. Data Store Access

Connection to MongoDB is achieved by using the Spring Data MongoDB library. This library hides various implementation details, and only URI definitions and credentials are required.

MongoDB is accessed through repository interfaces, which makes it easy to integrate MongoDB with Spring Boot. This repository interface inherits database operations such as finding with or without filters, sorting, counting, deleting, and inserting by default. In addition, functionality is not limited to these operations; thus, complex customized operations via annotations such as aggregation and query are also supported. The subscription service utilizes MongoDB’s aggregation feature for fetching data conditionally. So rather than fetching every document from MongoDB and processing it in Spring Boot, an aggregation pipeline containing several conditions returns only eligible records.

4.2.4. Exception Handling

Exception handling is a crucial part of robust software for various reasons. Program crashing, leak of sensitive data, and other unwanted behavior may be prevented with a proper way of handling exceptions. Fortunately, Spring Boot has built-in exception-handling mechanisms. Thus everything does not need to be built from the ground up.

Even though the default exception handler is capable of doing a fair job, sometimes customized handlers are the way to go since it provides better control over exception handling. Customized handlers in Spring Boot are created using annotation `@ControllerAdvice` or `@RestControllerAdvice`, allowing intercepting controller methods' return values. So instead of passing the default return value, we pass custom-created return values. Subscription Service implements four customized handlers for responding to user inputs verbosely. First, instead of returning the default unauthorized exception response when authentication fails, more information is returned to the user on what could be wrong, which helps the user or client application developer to figure out the problem. The second customized handler steps in when database queries do not succeed. Again, more information is returned to the user for convenience.

We also have a custom handler for cron expressions. It is easy to make a mistake in defining cron expressions, mainly since multiple formats of cron expressions exist. The quartz scheduling library in the message service makes use of cron expressions. Thus we need to validate the expression before storing it in the database. The last custom handler gives an accurate description of the user input validation. This information is required since subscription properties have constraints, which will most likely fail if the user does not get proper feedback on creating a subscription. The feedback also helps client application developers to create validity checks on the client side. For example, subscriptions need information about how it is triggered. For that purpose, the Subscription Service implements validity checking against predefined values before storing it in MongoDB. There are a couple of other validity checks, as well as type, not null, and regular expression checks.

4.2.5. Additional Libraries

Project Lombok provides tools in the form of annotations to write a lot less code by generating code at compile-time. The annotations we use in the Subscription Service are for generating java "setters" and "getters" automatically as well as constructors. Even though Lombok provides several other annotations, they tend to generate additional code we do not need. Therefore, these other annotations are not used in Subscription Service.

Another convenient library for Spring Boot is Swagger, which is often used as a tool for documenting APIs. As is usually the case with Spring Boot applications, Swagger functionality is used via annotations, and several annotations are available. However, documenting a single REST API endpoint can be done with little effort. For example, `@Operation` annotation takes in a description text about the endpoint, and `@ApiResponse` annotation takes in an HTTP response, HTTP code, and description. Once the annotations have been added, the documentation can be inspected in a web browser using the specified URL.

4.2.6. Containerization of Subscription Service

Subscription Service, as well as all other services that are part of the Watcher, will run in a Docker container. Containerization using Docker is a relatively straightforward process. The simplest instructions for creating the image include choosing the base image, copying the jar file, and command to start the application. However, there are aspects such as layers, base image, and total size that we need to consider when creating the Docker image from the Spring Boot application. For example, copying the jar file as a whole ("fat jar"), leads to one new layer, which is recreated after every change. In Subscription Service, we use the layer tool property to create a layered jar file to optimize the build process. Once a change is made to the codebase, only that layer is built instead of all. In the end, the final image is built using the so-called "multi-stage build," where the layers are extracted in the first stage and copied in the second stage. Another important thing in the Dockerfile is defining the specific base image tag to avoid surprises in the future. Docker provides an option always to use the latest image, which might seem handy, but the fact is that the newest image does not remain the same, thus may cause the build to fail or the application to crash, for instance.

We will create three containerized services with Spring Boot (the Subscription Service, the Data Processor, and the Message Service). Therefore, the process of containerization is somewhat similar and will not be covered again in each section.

4.3. Kafka Open Source Services

Kafka environment contains several services, which are described in chapter 2.6. This section covers how we implement Kafka Broker, Kafka

Connect and connectors[9][10], Schema Registry[11], and Zookeeper[12] using Docker and Docker Compose. Unlike the Spring Boot services, we will only apply configurations, so we place them under the same topic. However, they are open-source projects. Thus, further development is possible. Schema Registry and Zookeeper only require simple configurations to connect with other services, whereas Kafka Broker and Kafka Connect are highly configurable. Furthermore, Kafka connect accepts connector configurations via its REST API. The main tools and technologies for Kafka environment are presented in table 4. It is noteworthy that all these services are built from Docker images created by Confluent, even though multiple other images are available.

Kafka broker	Message broker between producers and consumers.
Kafka Connect	Stream data to Kafka broker from databases.
Schema Registry	Store message schemas for consumers.
Zookeeper	Manage Kafka broker and its data.
Debezium Oracle connector	Capture changes in Oracle database.
Debezium MongoDB connector	Capture changes in MongoDB.
Docker Compose	Simplify running of multiple Docker containers.

Table 4. The main technologies of the Kafka environment.

4.3.1. Services

As mentioned at the beginning of the section, neither Zookeeper nor Schema Registry requires complex configuration. The Zookeeper needs a port to be defined where it listens, while Schema Registry requires configurations for the hostname and one or several Kafka broker addresses. Figure 8 represents connections between services that utilize Schema Registry.

Kafka Broker can work with very minimal configurations as well. Only the Zookeeper's address and information where the broker is listening are required. However, there are numerous settings we can do to customize how the broker or even multiple brokers operate. In addition to the required options, there are some valuable configurations for us, such as self-defined

[9]https://debezium.io/documentation/reference/stable/connectors/oracle.html
[10]https://debezium.io/documentation/reference/stable/connectors/mongodb.html
[11]https://hub.docker.com/r/confluentinc/cp-schema-registry
[12]https://hub.docker.com/r/confluentinc/cp-zookeeper

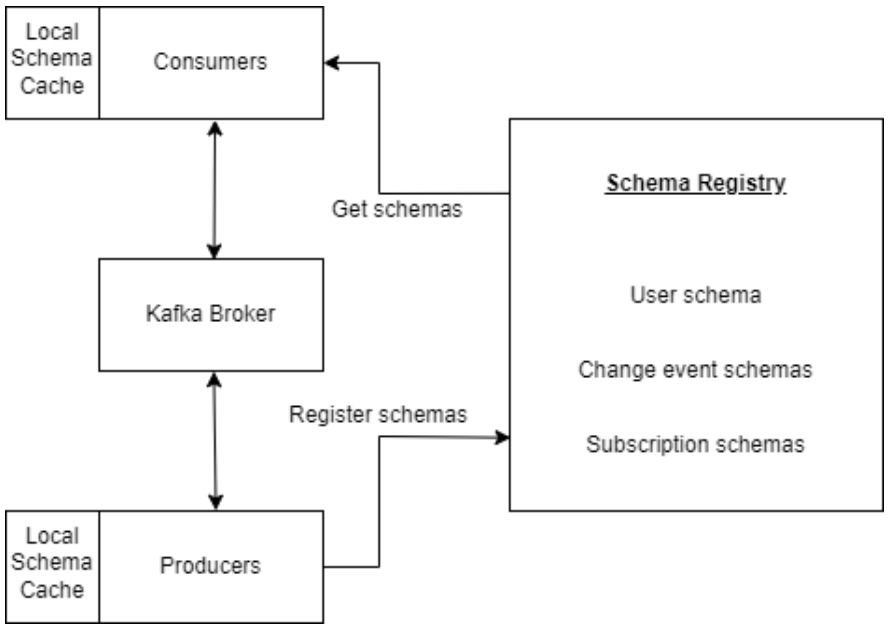

Figure 8. Avro schemas are available in Schema Registry after producers have registered them. Schemas are also stored in the local cache to speed up the process of using Avro data format.

id and offset topic replication factor, since we are running a single broker. Although, we will tune the values of these properties for production. However, many configurations do not need to be defined, as they have well-functioning default values automatically. Also, we need to create Kafka topics beforehand due to the nature of the Kafka Streams application we use in this project. It is because the Kafka Streams application tries to find topics we define in the application and cause an exception if they do not exist. To prevent the exception from occurring, we create Kafka-topics with a tool that comes with Confluent Kafka Broker by default.

Figure 9 The figure shows how producers write messages into Kafka topics comprising several partitions. Partitions can be configured while creating new topics.

While Zookeeper, Schema Registry, and Kafka Broker can work with minimal configuration, Kafka Connect needs nine settings in total. The settings are described below; key and value data converters are combined as well as storage topics due to the similar content.

- **Bootstrap servers:** Address of Kafka Broker or multiple brokers, including hostname and port number. Since we are running all the services in the same docker network, the hostname is the same as the container name.

- **Group id:** Identifier of the cluster group this Kafka Connect belongs to. However, only one Connect worker is running in a single cluster.

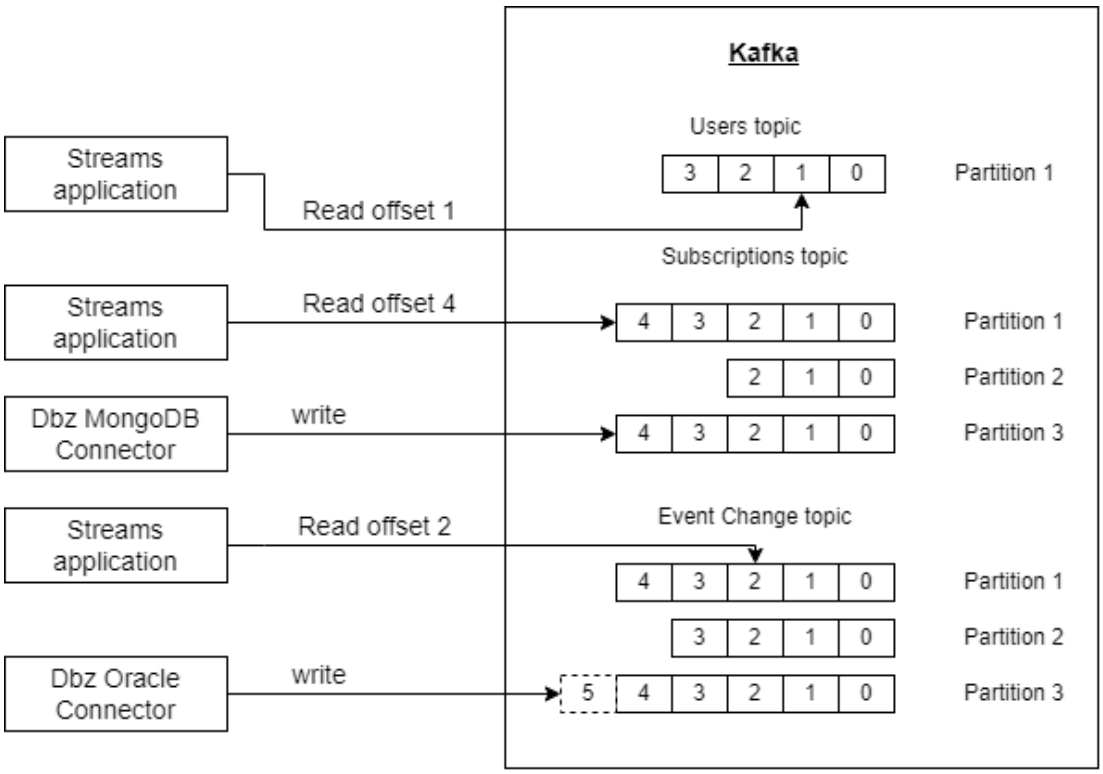

Figure 9. The figure presents an example of Connectors producing messages into topics. Subscriptions and event change topics contain three partitions due to immense data volume, while the user topic has only one partition. In addition, the Streams application is reading from arbitrary offsets of partitions.

Thus the importance of this configuration shows up less than in a multi-cluster environment.

- **Storage topics (3):** This is a combination of three similar settings. Configuration storage topic, offset storage topic, and status storage topic. A topic name must be assigned for configurations, offsets, and statuses within the same Kafka Connect group. Data related to these topics is stored behind the specified topics.

- **Data Converters (2):** Data in Kafka Broker is binary. Thus, brokers must convert the data from binary to binary depending on which direction the data goes. The data format for that operation is specified using key and value converter settings since each Kafka message contains a key and a value. As a value, we use Avro data format throughout the system. Therefore, the value-converter setting is an Avro converter. In contrast, the key-converter setting is set to string converter. In addition, to be able to use the Avro converter, the Schema Registry address also needs to be specified.

- **Rest advertised hostname:** Specified host name from which Kafka Connect can be reached.
- **Connect plugin path:** Location for plugins that Kafka Connect knows where to search. We use two connectors in this project. Thus we need to specify the path where the related files will be downloaded. More about connectors in the section. 4.3.2

We use Docker Compose to ease the workload since necessary configurations for all used services can be written in a file and start with one command without needing command line instructions. Docker Compose file sample presented in listing 4.3.1.

Listing 4.1. Docker compose file for running service of Kafka environment

```
version: "3.7"
  services:
    zookeeper:
      image: confluentinc/cp-zookeeper
      ports:
        - "2181:2181"
      ...
    broker:
      image: confluentinc/cp-kafka
      ports:
        - "29092:29092"
        ...
      ...
    connect:
      build:
        context: .
        dockerfile: confluentinc-custom-connect
      ports:
        - 8083:8083
      environment:
        CONNECT_BOOTSTRAP_SERVERS: "broker:29092"
        CONNECT_GROUP_ID: connect-cluster-group
        CONNECT_CONFIG_STORAGE_TOPIC: docker-connect-configs
        CONNECT_OFFSET_STORAGE_TOPIC: docker-connect-offsets
        CONNECT_STATUS_STORAGE_TOPIC: docker-connect-status
        CONNECT_ZOOKEEPER_CONNECT: "zookeeper:2181"
        CONNECT_KEY_CONVERTER:
        "org.apache.kafka.connect.storage.StringConverter"
        CONNECT_VALUE_CONVERTER: "io.confluent.connect.avro.AvroConverter"
        CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL:
        "http://schema-registry:8081"
        ...
      ...
    schema-registry:
      image: confluentinc/cp-schema-registry
      ports:
        - "8081:8081"
      environment:
```

```
      SCHEMA_REGISTRY_HOST_NAME: schema-registry
      SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: 'broker:29092'
      SCHEMA_REGISTRY_LISTENERS: http://0.0.0.0:8081
      ...
  ...
mongo1:
  build:
    context: .
    dockerfile: custom-mongo
  ports:
    - 27017:27017
  command: --replSet rs0
  restart: always
  ...
helper:
  build:
    context: .
    dockerfile: custom-helper-container
  command:
    - bash
    - -c
    - |
      # initialize connectors when the connect service is ready
      ...
      sleep infinity
  ...
networks:
  kafka-environment
```

Our services need to be launched in a specific order. However, Docker Compose version 3 does not support instruction keywords to make services dependent on other services, and we use Docker Compose version 3 since it is the latest. To solve this issue, we need to handle this in the application code. Fortunately, these services have a built-in feature to try to establish a connection until it is successful. However, connector configurations for Kafka Connect can be created once the Kafka Connect service is up. Also, they can be done by scripts or manually. In our case, we utilize helper containers to run scripts, which requires polling Kafka Connect and waiting for it to be ready. Furthermore, we configure one user-defined docker bridge network for these four services, which allows communication with container names instead of IP addresses 2.3.2.

4.3.2. Connectors

There is a data stream from the Oracle database and MongoDB into the system. For this purpose, we need a way to track changes happening in the database. Since we are using Kafka as the streaming platform, the best option is to use connectors for the job. Both connectors are created by Debezium, even though several other options are available as well.

Listing 4.2. Example Oracle connector configuration

```
{
    "name": "oracle-connector",
        "config": {
        "connector.class":"io.debezium.connector.oracle.
           OracleConnector",
        "database.hostname":"192.168.10.22",
        "database.port": "1521",
        "database.user": "debeziumUser",
        "database.password": "debeziumMinerPw",
        "database.dbname": "database",
        "table.include.list": "schema.table_one, schema.
           table_two",
        "database.server.name": "database_logical_name",
        "column.exlude.list": "schema.table_one.home_address",
        "database.history.kafka.bootstrap.servers" : "broker
           :29092",
        "database.history.kafka.topic": "event-changes.history
           ",
        "signal.data.collection": "database.DBZUSER.DBZ_SIGNAL"
}
```

The process of setting up the Oracle connector starts with preparing the database. We must first define the recovery file size and location. After this is done, supplemental log data need to be enabled for tables we are interested in and for the database. This operation allows additional columns to be logged into redo log files, which is required for log mining. The tables we are interested in contain information on users, items, and documents. The third step includes creating a user for log mining and granting various permissions required by the log miner.

The next task is connector configuration. What we need to do here is download the Oracle connector plugin archive and Java Database Connectivity driver for Oracle and place them into the Kafka Connect plugin path we specified in the Docker Compose file. However, it is inconvenient to do that manually, especially during development when it is most likely to be done several times. Therefore, we will automate the process by adding more instructions in the Dockerfile to create a Kafka Connect image.

Finally, when Kafka Connect is up, we are able to use its REST API to create connector configurations. An example HTTP request for creating a new Debezium Oracle connector configuration is presented in Figure 4.2

MongoDB has a feature called change streams, which is utilized by the Debezium MongoDB connector. The change streams feature works only with MongoDB sharded clusters or replica sets. Thus we need to use either

of those options with the connector as well. Sharded clusters work better in larger MongoDB setups. Therefore we chose to use replica sets, simply a group of MongoDB instances with the same data set. Another requirement for the database is to create a user with proper roles to read MongoDB oplog (operations log) from the admin database.

Debezium MongoDB connector is implemented similarly to the Oracle connector, which is downloading connector plugin files and adding them to the plugin path in Dockerfile. Also, the connector configuration is very similar to the Oracle connector configuration presented in Fig 4.2.

4.4. Kafka Streams

This section focuses on implementing the Kafka Streams application (Streams). We go through configurations, data processing, exception handling, and containerization.

Streams specializes in data processing, further, has the ability to consume from as well as produce messages to Kafka topics. In brief, Streams reads messages from multiple topics as a stream of events, does the data processing, and finally produces these input messages with modified data or creates completely new messages. One topic comprises one or multiple partitions with offset values that tell the location of each message in the partition queue. 2.6

In this project, we use Spring Boot to build Streams, even though it is not a requirement, and Java without any frameworks would also be sufficient. In table 5 is presented the main technologies we use in the Data Processor service.

Spring Boot	Framework for the Data Processor.
Maven	Building and managing the project.
RocksDB	The default key-value data store in Kafka Streams.
DSL	Domain-Specific Language for data processing in Kafka Streams.
Lombok	Reducing boiler plate code.
Avro libraries	Generating Avro deserializers and serializers.

Table 5. The main technologies of the Data Processor.

4.4.1. Service Initialization

As stated earlier, we use Avro data format throughout the system. Thus Streams need to be able to deserialize messages before using them. In the

Kafka ecosystem, we want to query Schema Registry REST API to acquire a schema since it is the service storing all the schemas. Once we have the schema, we can create a Java serializer and deserializer classes and start working with Avro. However, the Apache Avro compiler is able to construct these classes for us automatically based on the Avro file. The compiler is imported as a Maven dependency and runs at application start-up.

There are two mandatory configuration parameters for Streams; application id and broker addresses. In addition, Streams accepts configurations such as customized key and value converter classes as well as the address of Schema Registry. In our case, these additional configurations need to be specified in order to use Avro. We also need to appoint error handler classes, since the default ones are not eligible, and set configuration parameters for RocksDB key-value data stores.

We need three configuration objects in total due to how this system is built. The first object is for a stream for processing user data, which differs from the subscription configuration object in how it utilizes RocksDB and when the streams should start. Thus we need to set at least appropriate memory parameters for both streams and disallow automatic start-up for subscription streams. The third configuration object is for event change streams, and this stream does not use RocksDB. An even more important parameter to set is the offset, which specifies the location where a stream starts reading. We need all users and subscriptions from the databases. Hence we need to start reading from the offset zero, while event changes that happened in the past do not bring any additional value. Therefore, the third configuration object is set to start reading Kafka messages from the current offset.

Each Avro message is deserialized or serialized by a provided Serde (serialize-deserialize) class. In fact, every Streams application must provide Serdes, and it does not limit to Avro format only. Every time a message is read from a Kafka topic, a key and value must be deserialized. Since we use strings as keys, we can use built-in Serdes. However, the value of a message is in Avro. Thus it is required to come up with customized Serdes. Fortunately, our generated Avro serializer-deserializer classes can be used for creating custom Serdes.

We need five RocksDB data stores to carry out all required tasks; four for storing different types of subscriptions and one for users. RocksDB is the default for Streams, and we do not see any benefits of changing it. There are two options to operate with the data store; Processor API or DSL, which is built on top of the Processor API. DSL simplifies several things for initializing, storing, or reading from the store. Although, DSL functionality is more restricted and does not allow other than reading operations. However, DSL is very much capable of performing all the necessary tasks to manage our data. By using DSL, among other things, we are able to materialize users and subscriptions into the stores and update

them automatically based on the key. This mechanism precludes having multiple versions of the same object.

The reason for storing users and subscriptions in RocksDB is to speed up the data processing since each time a new change event arrives, user and subscription data is available instantly. Another way of doing this would be querying subscription data from MongoDB and Oracle databases, which is much slower than RocksDB. Furthermore, Kafka is an event streaming platform with powerful features available to us.

We talked about multiple Streams configurations earlier in this section. One advantage of having separate configurations for each subject: users, subscriptions, and change events, is the possibility to start up each stream manually and further initialize data stores in the desired order. The following list describes the desired order in this case.

1. **Users:** Stream starts after the Spring Boot application is up and running, followed by populating the store.

2. **Subscriptions:** Starts right after the Users store is populated. The Users store is populated first before storing any subscriptions, and each subscription will be enriched with email addresses available in the Users store.

3. **Events:** As soon as Subscription stores are populated, we can let the event change stream come in and start processing data.

4.4.2. Data Processing

Continuous data flows into the Streams application, which is either utilized to process incoming data or modified by other incoming data before producing it to a Kafka topic. Here we present topics the Streams is listening to and all the steps of heavy processing, which ends up either storing or discarding messages or producing them into a Kafka topic.

Streams DSL has three general concepts for processing data: `KStream`, `KTable`, and `GlobalKTable`. However, we will not use `GlobalKTable` in this project. `KStream` is a stream of self-contained data, while `KTable` data is an update to existing data. `KTable` is used for user and subscription data since we are only interested in the latest value. Change events are all independent, and consequently, we use `KStream`.

The user stream does not do much processing but is a crucial part of the processing overall. First, we filter out all inactive users, after which we prune unnecessary user data. Finally, users will be stored in RocksDB. If user data changes, Streams will receive the update, and the new user data is used to replace the old one.

The subscription streams comprise four streams in total. In general, all subscriptions go through the same process of data enrichment. However,

if the subscription delivery type is email, it will be produced into a Kafka topic and stored in RocksDB. The reason for this is that Message Service needs the information to be able to create scheduled events.

Processing change events are by far the most complex task. The whole process is depicted in Figure 10. First, we need to filter out all change events that do not meet the initial requirements, such as changes in private objects or those containing an object code that cannot be found in any subscriptions. Then we go through all filters of all subscriptions to narrow down the results. Finally, we are able to take the original event change and create a new object without unnecessary properties, after which we add subscriptions to that object. If no subscriptions are found, the execution of the program ends there. However, before producing the new change event object as an email event or a push event, all subscribers without proper permissions are removed.

4.4.3. Exception Handling

Continuous data flows to and from Streams, increasing the chance of encountering an exception or failure. If we add the complexity of data processing and stateful functionality, it is impossible to avoid exceptions. Therefore, this phase of the development process is required to be done with care.

Basically, there are three main categories for errors: Consumer exceptions, processing exceptions, and producer exceptions [44].

Consumer exceptions relate to a situation at data entry. The default exception handler for these types of exceptions is the `DeserializationExceptionHandler` interface, which is set to log and fail every time an error occurs. Log and fail sort of behavior we want to change. Thus we have two options; change the default handler or create a custom handler. The solution for us is to use the default handler since it provides an easy way to change the behavior as we want. All we need to do is replace the default handler configuration `LogAndFailExceptionHandler` with `LogAndContinueExceptionHandler`, which results in the application continuing even if a consumer exception occurs. Typical exceptions, in this case, includes network and deserialization exceptions.

The other error-prone step is right before messages are produced. This time the default handler is `ProductionExceptionHandler`, which does not know any other outcomes than failure. Therefore, the customized exception handler we created does not fail but logs the exception and continues. In addition to just logging exceptions, the application notifies us immediately, allowing us to do the necessary fixing.

Data processing exceptions appear between consumer and producer exceptions. Streams provides `StreamsUncaughtExceptionHandler` for dealing with exceptions during data processing. We can replace the current

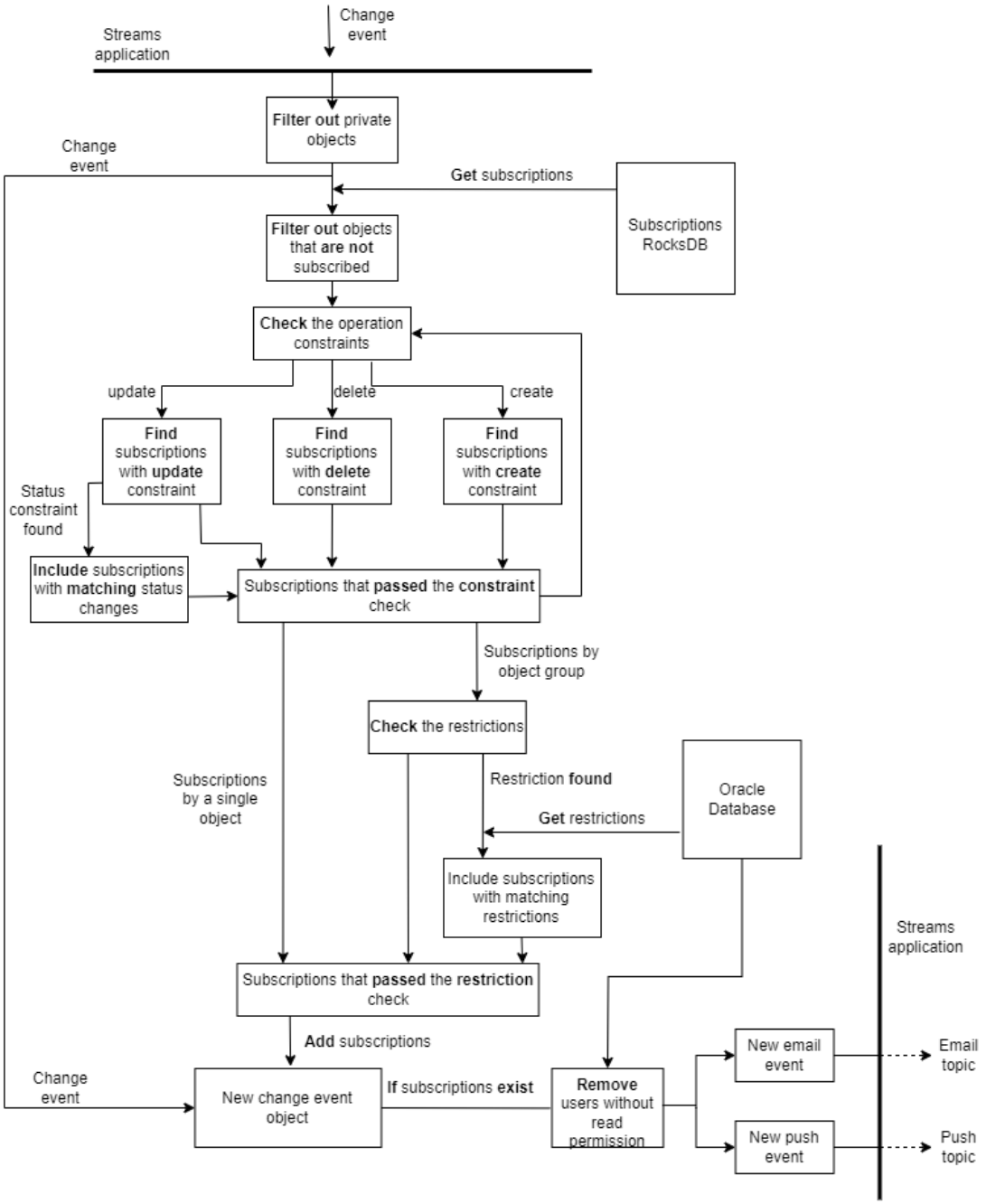

Figure 10. Processing event changes

thread and shut down the current instance or all instances. To avoid the instance shutting down, we first create a new thread. However, if the exception persists, the instance is then allowed to shut down. It works in a way that the thread is replaced repeatedly until some predetermined maximum number of failures per time frame is exceeded. After the limit is hit, the instance will shut down.

4.5. Message Service

The Message service is the last one, literally as well as figuratively. In this chapter, we present the implementation of the Kafka message listeners, Quartz scheduled tasks[13], and email, as well as push events that will be sent to users. Since we can point out two branches of data flow, email and push events, they are covered one by one rather than mixed together by phase. To stay consistent, and in addition to the arguments presented earlier, we chose Spring Boot as the framework for the service. Table 5 below shows the main technologies we use in the Message service.

Spring Boot	Framework for the Subscription Service.
Maven	Building and managing the project.
Lombok	Reducing boiler plate code.
Quartz	Scheduling tasks.
JavaMail	Sending emails.
H2	Database for scheduled tasks and processed Oracle database changes that the scheduler uses.
SSE	Sending push notifications to clients.
Reactor Kafka	Non-blocking Kafka consumer for retrieving messages from Kafka topic and make them available for SSE component in the Message Service.
Kafka consumer	Consuming new or updated subscriptions as well as email events from Kafka topic.
Avro libraries	Generating Avro deserializers.

Table 6. The main technologies of the Message Service.

4.5.1. Subscription Events

First, we need to set everything up before letting the stream of events flow in. The message service is a regular Kafka consumer that listens to topics for new data appearing. Avro data format, used throughout the system, must also be handled in the Message service. Thus, we must assign an Avro deserializer class and addresses of the Kafka broker and the Schema Registry to the subscription consumer. Furthermore, we need to specify an offset and tell it to read from the beginning of the subscription message queue.

Once we have the configuration object, we can create a consumer to listen to new subscription events. This listener is quite simple considering the functionality since as soon as a new subscription event appears, it will be

[13]http://www.quartz-scheduler.org/

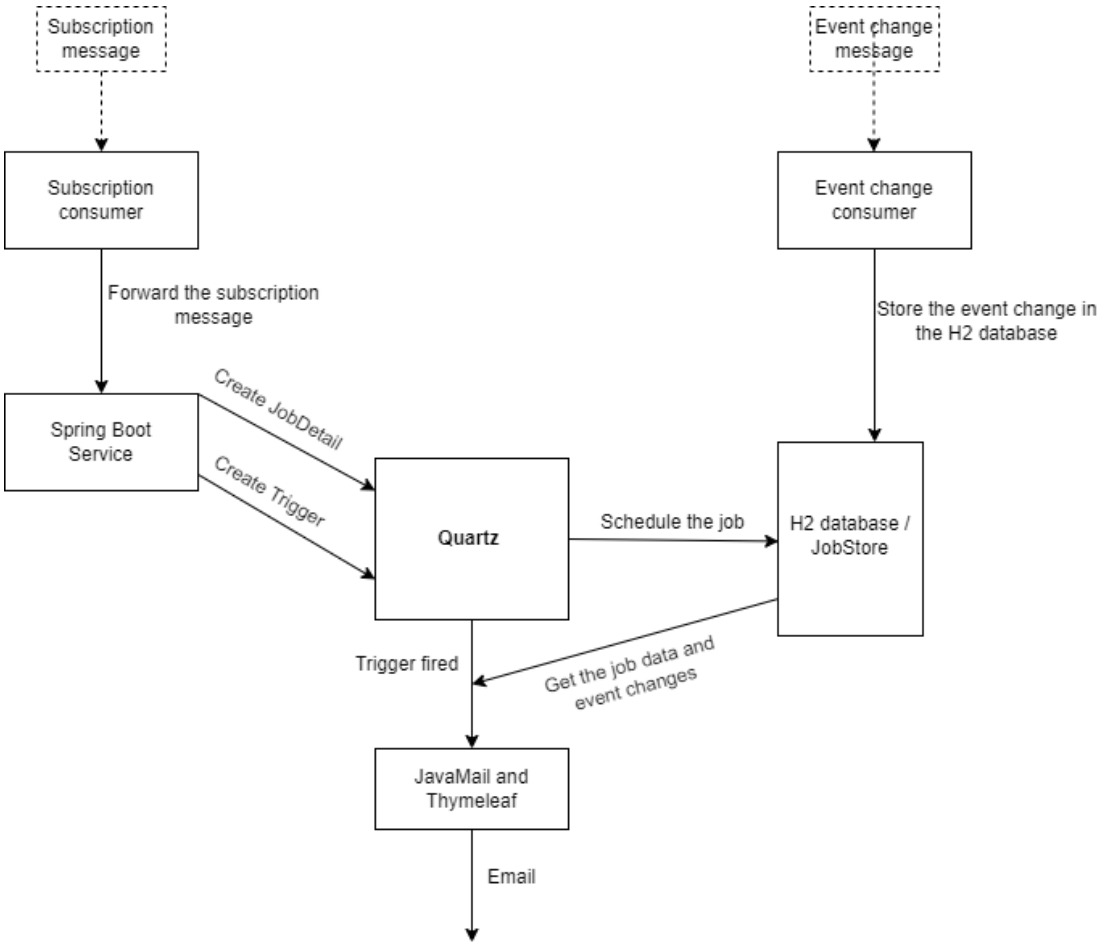

Figure 11. The figure presents incoming messages: subscriptions and event changes, going through the steps of scheduling, storing, and mailing.

forwarded to the Quartz scheduler service, which extracts the data and creates a new scheduled task using that data.

Before we are able to use the Quartz scheduler, there are some configuration options to consider. First, we specify and configure the data source for Quartz. As stated in the chapter 3, we use the H2 database. Even though one of the criteria was that the data must be persisted, we used volatile storing options during the development. Another essential thing is to initialize the database with tables for Quartz. Fortunately, all the table definitions are created beforehand by Quartz, and we need to apply them only. Since we are using Spring Boot, we should also utilize the functionality it provides throughout the application instead of using Quartz with straight-up Java. To do so, we need to let the Spring Inversion of Control container take care of the life cycle and build the code in a way that does not mix Spring Boot style with the regular Quartz commands. The combination of these two will most likely lead to unwanted behavior.

Despite the capabilities of Quartz customization, the process of scheduling a task is relatively straightforward, having three elements:

specifying job details and triggers, followed by sending those to the Quartz scheduler. In the JobDetail object, we will add task-specific configurations such as the task to perform and the name of the task. Furthermore, each task contains a data map containing information such as email addresses and subscription IDs received from the subscription listener. For the trigger object, we specify the time when a task is triggered and the misfire instructions. Even though selecting the triggering time from various options is possible, we find the cron expression to suit the best for the Watcher. Like the information for the JobDetail object, the cron expression is available in the same subscription message.

As we recall from the Subscription service implementation, all subscriptions contain a list of subscribers, and the list is modifiable, even though the subscriptions are not. If the list of subscribers is modified, it needs to be brought into the message service and replaced with the old list of subscribers. All scheduled events contain an id, which is the key to whether a new task should be scheduled or modify an existing one. If it is the latter option, the right task can be found in the H2 database, and further, write the correct values in it.

A task class specified for the `JobDetail` object is called as soon as the job is triggered. The specified class tries to find any event changes matching the triggered subscription job. In case even one event change exists, the execution will continue; otherwise, the job is done but the scheduler remains to wait for the next trigger time.

Event changes and the information in the job data map will be combined and sent to subscribers using the JavaMail[14] API and Thymeleaf[15] templating engine. With Thymeleaf, we can create beautiful emails containing dynamic data, while JavaMail provides an easy solution for sending emails to multiple subscribers at once.

Figure 11 presents the steps required before any emails are sent.

4.5.2. Email Events

Email events consumer has the same set of configurations as subscription consumer, excluding identification details. Thus we can utilize the same configuration object we created earlier. Event consumer listens to the topic where all event changes are published and stores them in the H2 database for later use. Another option for consuming event changes included constant resuming and pausing of the consumer to avoid storing them in the database. However, Kafka is designed for a continuous stream of data. Thus we chose to keep the stream open and use a temporary data store.

[14] https://javaee.github.io/javamail/
[15] https://www.thymeleaf.org/

The lifespan of an event change in the Message service, including email as well as push events, starts when they are consumed from Kafka and ends when they are removed from the database due to their predetermined expiration date. The expiration date is set in order to save storage space but also because we will not gain anything by not removing the data.

4.5.3. Push Events

Push events make use of the Reactor Kafka API, which provides an asynchronous, functional way to build the pipeline for delivering event change messages to subscribers. Using the Reactor, we omit complicated tasks such as monitoring memory usage and limiting the data flow. Despite the initial plan of using WebSockets to push messages to subscribers, we ended up creating a controller class that implements the SSE way of delivering the messages. The reason is how well the SSE plays together with the reactive Kafka without adding unnecessary complexity.

For subscribers to be able to start receiving even change messages via SSE, they need to make an HTTP request, including a security token, to an endpoint with a request parameter containing the user code. The only difference with common JSON-type HTTP requests is the Multipurpose Internet Mail Extensions (MIME type) of SSE, event-stream. Once the connection is established, Reactive Kafka starts handling messages and filters them based on the user codes received from the clients. Figure 12 presents the interaction between a client application and the Reactive Kafka consumer

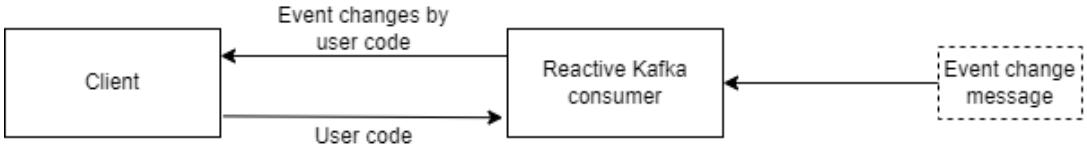

Figure 12. Connection between a client application and The Reactive Kafka consumer is established to start receiving event changes.

4.6. Application

Finally, all the necessary components are built, configured, and ready to work together. However, launching the whole system requires more work than we wish, which is to put all services in motion with one command. Therefore, we need a Docker Compose file and scripts to automate the process. (See figures 4.3.1 and 4.2.) Docker Compose will contain instructions to start up all the services: the Subscription service, MongoDB,

Kafka environment, the Data Processor, and the Message service. We will also create proper Docker network bridges for isolation. The shell scripts are for creating Kafka topics and initializing MongoDB and Oracle connectors. In addition, we use the Javascript programming language to set up replica sets for MongoDB and initialize collections for subscriptions. The execution of these scripts is done during the start-up by a helper container. The helper container is built on top of an Ubuntu image containing a large set of tools for managing the start-up process. Furthermore, the helper container will not shut down as soon as the system is up but will remain to assist with monitoring and debugging.

5. RESULTS

This chapter covers how we succeeded with the requirements described in chapter 3. In addition, we compare the working principles and technologies of the system built during this project and the old system, LSS.

5.1. Success with the Initial Requirements

In this section, we discuss the initial requirements, functional 3.2.1 as well as technical 3.2.2, and go through how well the system is able to fulfill those requirements. First, we will focus on the functional part, followed by the technical requirements.

Table 7. Summary table for the functional requirements.

FR1	Informing	✓
FR2	Edit subscriptions	✓
FR3	Create a subscription	✓
FR4	Browse subscriptions	✓
FR5	Create filters	✓
FR6	Admin features	X
FR7	Emails	✓
FR8	Data migration	X

5.1.1. Functional Requirements

The functional requirements are summarized in table 7. The first requirement we set comprises informing subscribers in two ways; via email or as a push notification **(FR1)**. Even though we decided to change the design regarding how the push notifications are delivered, we made them available using SSE. In the end, by performing an HTTP request on a specific endpoint that carries an authorization token in the header and a user code as a parameter, the Message service will push the event changes to the client application as they appear. The data is sent as JSON, leaving the representation up to the client. The message service is also capable of sending emails using the JavaMail library and the Thymeleaf templating engine to create stunning emails.

(FR2) states that a user needs to be able to subscribe or unsubscribe to an existing subscription. The REST API we built in the Subscription service offers an endpoint for modifying subscriptions. However, the list of subscribers is the only modifiable property since there is no need for further

adjustments. As a matter of fact, allowing subscriptions to be changed later would cause some detrimental side effects. Subscriptions can be made public, and other subscribers expect to get what they have subscribed to. Thus changing the delivery type or filters, for instance, would completely change the subscription base.

In addition to **(FR2)**, users can create new subscriptions **(FR3)** in case none of the existing subscriptions provide the desired information. The REST API in the subscription service implements endpoints that can be used to retrieve subscriptions if they are marked as public **(FR4)**. Private subscriptions are available only for the user who created them.

Filtering **(FR5)** does not specify explicitly the filters we need to provide. However, we implemented a variety of them due to the vast amount of database changes which would further cause a lot of unwanted data traffic. Fundamentally, the goal is to provide a small amount of valuable information that allows people to work more efficiently instead of flooding them with futile data. The system architecture will enable us to add more usable filters later on. Currently, we have covered subscriptions by object code or object group, relations with other objects, specific status changes, deletion, version updates, and object creation. This collection of filters enables subscriptions with very detailed data.

Users with admin privileges should be able to manage subscriptions for other users as well **(FR6)**. However, the feature is not supported currently due to the time constraints set for the project. This missing feature does not prevent the use of the Watcher, though, since it is considered an extra feature rather than blocking. Fortunately, the clean structure of the API allows adding this feature later on without breaking anything or forcing us to change existing code.

(FR7) has some overlapping with **(FR1)**, but the difference is dynamically selected email templates. Customers may prefer some templates over others, or they might want to get their logos added to the template. Thus each client will have a different set of templates. The template is chosen based on the change event, which could be "new version," "object deleted," or "status changed," for example.

The last functional requirement states that subscriptions created over the years since the LSS was first published should be usable in the Watcher **(FR8)**. First, this is the last step before the system can be delivered to a customer. Second, it will not be a core part of the Watcher. Thus, implementation of this feature is put off until we are ready to go production.

5.1.2. Technical Requirements

Technical requirements are a combination of common characteristics of quality software defined in ISO/IEC 25010:2011 [45] as well as application-

Table 8. Summary table for the technical requirements.

TR1	proper feedback	✓
TR2	reliability	✓
TR3	fault tolerance	✓
TR4	safety	✓
TR5	scalability	✓
TR6	maintainability	✓
TR7	SRP	✓
TR8	various platforms	✓
TR8	Persistence	✓

specific features we found important. The summary is presented in table 8

We define an application to be robust if properties such as proper feedback **(TR1)**, reliability **(TR2)**, fault tolerance **(TR3)**, safety **(TR4)**, scalability **(TR5)**, and maintainability **(TR6)** exist. However, we tested these requirements by running all the services alone as well as together. Inputs to the REST API in the Subscription Service were created using Postman[16]. While inputs to the PDM system were created using existing scripts we had to generate event changes. Then we tested a common issue in distributed systems, network exception, to draw out services that may not be able to recover. We also tried to generate inputs in different volumes to point out possible bottlenecks in the system. After tests, the application was left alone for a couple of days to find out if we ran into an exception.

We created all possible combinations we came up with in order to test the Subscription Service, resulting in expected feedback **(TR1)**. The feedback included the correct HTTP response status codes and instructions to reform the request and make the next input pass the validation. Logging throughout the system was implemented to provide developers valuable feedback, including significant events and exceptions such as new scheduled tasks and a key-value store unavailable.

User input validation in the Subscription Service is a crucial part of the overall reliability **(TR2)** since a wrong kind of input most likely affects data processing negatively. Thus we cannot allow half measure with the input validation. For example, neither incorrect filters and cron expressions nor unidentified usernames and unauthorized users did not pass the validation. Then we tested the accuracy of data delivery by ensuring all event changes were delivered via email and SSE. The results turned out to be only almost as expected. And some of the event changes were produced two times instead of once. Also, in theory, the Quartz scheduler task would be delayed or missed if it cannot handle tasks due to misconfigured number of threads

[16]https://www.postman.com/

or otherwise misconfigured policies. However, we did not encounter such behavior. The Watcher comprises several services which require a particular start-up order at the system level. After multiple launches, we discovered that everything works flawlessly if we start from a clean slate. Still, if there were any complications while a container shut down last time, it resulted in a failure occasionally. In addition to the system-level start-up order, the Data Processor contains elements that also need a specific order at start-up. However, we did not encounter exceptions due to the service's ability to constantly check that each phase is completed after moving up to the next one.

Neither the Subscription Service nor the Message Service indicated any non-recoverable exceptions. During the testing period, the Subscription Service handled all exceptions as described above in the feedback section, as well as the Message Service. The Data Processor, however, raised a deserialization exception, but the issue was quickly fixed and did not appear after that. The only concern regarding the fault tolerance **(TR3)** of the Watcher comes from the Oracle connector, which does not seem to be able to recover after a networking error without manually restarting the connector.

None of the endpoints of the Subscription Service or the Message Service were accessible without a valid authentication token, as expected. All the services besides these two are isolated and inside a Docker network, which is not accessible outside the network and increases the level of security **(TR4)**. A noteworthy fact is that the Watcher will not be exposed to the Internet directly but will be available behind all the security walls clients have set for their companies.

We tested two dimensions of scalability **(TR5)**. The first indicates how well the system can adjust to alternating data traffic. The other one comprises factors that define how well the system is able to adapt to future changes. The latter also falls into maintainability, which is why we will cover that below in the maintainability section. After exhausting testing, we were able to narrow down possible bottlenecks into one, the Data Processor. Even though the system handled increasing traffic well, marginal delays could be seen in certain filtering in the Data Processor. Nonetheless, the delay originates from SQL queries that contain multiple steps, such as checking several object relations. However, tests were performed with only one Data Processor instance, leaving us a fair amount of space to scale up horizontally. Kafka broker can handle trillions of messages in a day though [30]. Thus we did not find it practical to try to reach that limit ourselves.

Code quality and language-specific conventions, file structure, documentation, and system architecture are the factors we considered to impact the overall maintainability **(TR6)**. While there are no strict rules for building Spring Boot applications and event-based systems, all factors above follow recommendations or general guidelines as much as possible. Code neatness is achieved by applying an IntelliJ IDEA style

checker, which helps to stay consistent with the code format. File structure follows the Layered pattern 4.2.1, which forced us to structure the content in a modular, easy to understand by other developers, thus increasing the maintainability. With the system architecture, we needed to follow more than one pattern or principle to achieve the desired results. Figure 13 presents the complete architecture. Documentation is done with Javadoc and Swagger. Even though we could have used Javadoc only throughout the whole system, the company preference has been Swagger for REST APIs, and we want to stay consistent. Thus, we use Javadoc to document Java classes and methods placed in all other locations than in the controller directory in the Subscription service.

The rest three requirements **(TR7)**, **(TR8)**, and **(TR9)** are application-specific and thus do not necessarily increase the level of robustness.

(TR7) defines that each of the seven services has one main reason for their existence. Even though the names of these services are very self-describing, the list below describes the purpose of each Spring Boot service more accurately. Services for the Kafka environment are explained in chapter 2.6.

1. **Subscription service:** A REST API for creating, modifying, and fetching subscriptions.

2. **Data processor:** Modify and filter messages available in Kafka topics and produce these processed messages into Kafka topics.

3. **Message service:** Deliver processed data to subscribers via scheduled email or SSE

(TR8), Cross-platform challenges are solved with Docker since docker containers can be run in all our client's environments. Another convenient outcome of using Docker is that we can avoid extra work with installation due to existing Docker solutions.

(TR9), Persistence is achieved with several data stores: MongoDB, RocksDB, Kafka log, and H2. Although, the only relevant persistent data store is MongoDB, which is used to store subscription information. The rest could be compared to regular caching since it does not matter if the data is lost because these data stores get their data from MongoDB. Therefore, we tested how the data is recovered after the services crash, and the results were quite promising. The data processor and the Message service recover their data from Kafka topics automatically after crashing, and Debezium connectors were able to recover the data of Kafka logs with their snapshot feature.

5.2. Comparison to Predecessor

Both services work in very different ways; of course, the LSS was built at the turn of the millennium, making the comparison somewhat tricky. Also, the number of services in the old system is just one, while the new system comprises seven services in total, forming a distributed system. SRP and the Kafka environment explain the considerable difference. To follow SRP, each service should have only one responsibility. Therefore, the new system has a service for users to interact with the system, a service for data processing, and a service for message delivery. Furthermore, the Kafka environment consists of four services. The old system does not separate the first three responsibilities and does not use Kafka or event-streaming at all.

First, we go through the basics of how both systems work and what it takes to get them in the state where we can use them, after which we discuss technologies choices.

5.2.1. Working Principles

From the user's perspective, there are no significant differences in how subscriptions are created. They both accept HTTP requests for working with subscriptions. However, the rest is constructed in a very different way. In the Watcher, new or modified subscriptions are sent to the Data Processor service via Kafka Connect and Kafka broker and then back to Kafka broker, from where the Message service retrieves the enriched data. The old system, however, contains one service for scheduling and the Oracle database for storing subscription information. Most of the logic is taken care of with Oracle's functions and procedures. The new system works with a continuous stream of unbounded data. Or in other words, it will not fall asleep for a certain period of time but keeps working non-stop for better performance. The old system has a predefined loop time, 10 minutes, for instance, which means that once in ten minutes, it checks if event changes are waiting to be delivered. The new system delivers event changes via appealing emails or SSE, while the old system is able to send emails only. However, push notifications were supported years ago, but the feature has been deprecated, and the delivery was never in real-time.

5.2.2. Installation Process

The installation process with a script is relatively straightforward for both systems. The main difference is that some of the pieces in the old system are embedded into the main application, the PDM system. Thus they come with the standard installation process and cannot be removed safely. While

the new system is an entirely separate application and only streams event changes from the PDM system or, in some cases, queries the database.

5.2.3. Technologies

Technology stacks have almost nothing in common except they use Java mainly. Web services for the old system are implemented directly with Java servlets and Java Server Pages. In contrast, the new system uses Spring Boot and provides data in JSON format to client applications. Even though the new system gets all the change events from Oracle, the primary database is still MongoDB, where all the subscriptions are stored. The old system works with the Oracle database only. Since we use the Debezium connector to get the data out from the Oracle database, we can no longer work with the Oracle XP edition. XP edition is a lightweight version compared to EE and SE versions and does not support log mining. Comparing the rest of the technologies would be difficult because we have two completely different systems.

6. EVALUATION AND DISCUSSION

This chapter discusses the design, implementation, results, and future. The design contains a detailed thought process of how the final architecture was derived from a situation with nothing but functional requirements for the system. In the implementation, we cover encountered challenges and how we overcame those, as well as explain why the design panned out differently than planned. The result section includes a summary of the overall success of the project. Lastly, in the future section, we bring up some observations, from our viewpoint, that should be addressed in the future.

6.1. Design

The first step of the design was to define the problem: how to capture changes in the Oracle database and deliver them to users based on their preferences. We had two options for capturing changes: database triggers and log mining. However, one of the goals we aimed for was to separate this new Watcher system from the PDM system as much as possible and avoid using anything that requires changing the codebase of the PDM system. Log mining would bring us much closer to that goal; therefore, the choice is clear at that point. The second half of the problem includes working with user preferences; thus, interaction with the system should be allowed somehow. We had one option to carry this out, and we were convinced it was REST API.

Then we pointed out most necessary components that the system may require, and we discovered that at least a database, task scheduler, and email client would be crucial. Also, there has to be a way to push real-time events to the users. At this point, we had a monolith that received changes from the Oracle database, processed them, and sent the data to users via email or some push mechanism. The monolith would also contain a REST API and access to another database for storing user preferences.

The system architecture starts to take shape after discovering Apache Kafka and Debezium Oracle connector and how well it plays together with our goal. Although there are several options for capturing changes in the Oracle database, we had to go through them all to justify why we chose Kafka. To put it simply, this setup offers open-source tools to build a well-scalable system that fulfills our requirements completely. Furthermore, Kafka has a large online community for help as well as it is used and tested by some of the biggest companies in the world. Therefore, we do not expect the support to end in the near future.

The data format used throughout the system is a choice, even though the recommendation for Kafka by Confluent is Avro [46]. We have not worked with Avro before, but this is going to change now since the creators of

Kafka have a solid recommendation to use that. Thus, without exhaustive research on that matter, we discarded the idea of using familiar JSON.

While Kafka is able to perform several tasks, it is fundamentally an event-streaming platform, and now we need to start looking more toward distributed systems and event-driven architectures over monoliths and simple request-response models. So instead of polling changes, the current plan is to keep the stream of events from the Oracle database open and store all the events in a different database, MongoDB. We figured out this would create some bottlenecks since the data processing happens in chunks. Also, it felt quite an inefficient way to carry this out. Naturally, the next option to consider was to route the traffic somewhere for data processing before storing it in MongoDB.

One major decision was how and where the data processing is done, which led us to consider three completely different options: transformers in the Debezium connector, a set of methods in an existing Spring Boot application, or Kafka Streams application. We decided to go with Kafka Streams, separate all data processing and call into play its power, even though we were not aware of all its capabilities at the moment.

Yet, there was one thing to do before we were satisfied with the architecture. Since we already have multiple services responsible for a single task, we decided to go further and separate the REST API from message delivery and job scheduling to achieve cleaner architecture. Features implemented in the Message Service and the Subscription service have nothing in common whatsoever, and thus they should not be implemented in the same service according to the SRP. By separating all responsibilities, including data processing, we also aimed for better maintainability. However, at this point, we had all the services as we wanted 5, but the details are far from ready.

In terms of the architecture details, the most time-consuming was to go through all the different scenarios related to the data flow and storage. We were trying to get things rolling with ideas such as storing all data in MongoDB, caching data in the services, need for another database, using Kafka log as a database, and keeping consumers in the Message Service running continuously or only when a scheduler job is triggered. There were numerous things to consider. After struggling, we were confident with the final design (See chapter 3 for details).

All the technologies we use can be justified, especially those with significant roles such as Kafka, Docker, and Spring Boot, nevertheless, only one missing feature could have prevented using some otherwise great technology. Thus, we required the technology to be widely used, which also often indicates support for the developing phase as well as hopefully for the following years or decades.

The design was purely theoretical. In the beginning, we only had little, if any, experience with the technologies used during the project. For that reason, it felt obvious to build a test environment that only focuses on the

big picture omitting all details. This solved many issues in the early stage, which would have been more expensive later in terms of time. Despite the great design, we were still suspicious that the project was too complex to build and can we get things done as planned. Another thing was causing a headache; should or could we simplify the architecture since the system comprises seven services in total, whilst the LSS is only one service?

The design comes with downsides as well. Technologies and complex architecture might intimidate at first, and even though the design promotes maintainability, it applies only if the developers take some time first to familiarize themselves with the project. Also, a lightweight Oracle XP does not support log mining.

6.2. Implementation

Design and implementation are two different things, which we experienced several times while polishing up some features or even getting them to work. Avro data format, for instance, is the best option for Kafka-based applications; nevertheless, we found using it overwhelming at first. The whole process includes the following:

1. Generating Java code from Avro files.
2. Configuring Kafka consumers and producers to use the specific Avro serializer.
3. Creating Kafka Streams Serdes (serializer-deserializer) methods.

In addition, sometimes, we had to query the Schema Registry in order to get the proper Avro file. Despite these quite straightforward steps, a deserialization exception was thrown by some of the services once in a while. However, once we were able to push through these steps successfully, Avro began to show glimpses of its superiority.

The list of encountered challenges is quite long. However, most of them relate to data processing somehow. The Data Processor Service is built with Spring Boot, but most online help is available as pure Java code. Thus we found ourselves trying to convert the Java way of doing things to Spring Boot's way. This includes access to the key-value stores and configurations of multiple streams as well as the start-up order of those streams in the Kafka Streams application. Also, the Streams do not allow starting from a clean slate without extra work since it requires deleting key-value stores in the file system and removing Streams-related information from the Kafka broker. Fortunately, the Kafka broker by Confluent has a tool for removing the information.

Finding the final configurations for Kafka broker and Kafka Connect, as well as for Debezium connectors, was a lot of work. Even though the

documentations are pretty clear, the best describing method for the process is "trial and error." In addition, the Oracle connector stopped working once due to misconfigured recovery file destination size. This is not an issue, but we would expect some feedback other than just denying queries.

Implementation of the Message Service, as well as the Subscription Service, was relatively easy compared to the Data Processor Service. Only minor challenges were encountered, such as injecting Spring beans in the Quartz context and streaming with Reactive Kafka.

We could say that the overall design was a success since only three things in the implementation do not match the design. First, we planned to use WebSockets for real-time communication between the Message Service and client applications. However, we made a last-minute change, and the communication is implemented with SSE due to its simplicity. However, we are not sacrificing anything with this trade; all features we need can be fully accomplished with SSE as well. Secondly, we wanted to avoid database queries in the Data Processor since they have a significant impact on the speed. Unfortunately, business-related concerns left us with no options other than occasional Oracle database queries. Thirdly, MongoDB is for the subscription data only, even though we planned storing the user data there as well.

6.3. Results

Despite the enormous amount of work we put into this, we were not able to complete all the functional requirements; admin features **(FR6)** and data migration **(FR8)**. Admin features, creating subscriptions for other users, are only an extension to the normal process and thus were not prioritized high at this point. Data migration, however, becomes crucial after we have been honing the system for months and the project state changes from in-development to production-ready. We are also a little skeptical about the great testing results since the system passed almost them all. Even though the system should not indicate any abnormal behavior, we still expected more of that behavior due to a relatively small amount of testing before executing the official tests.

We also expected the results to indicate an improvement in the LSS. It is also an opinion, but the new version provides better performance, scalability, maintainability, and functionality. It is also built with state-of-the-art technologies and is easier to bring into use. It is only speculation, but there is a good chance that these new technologies are more straightforward to embrace than legacy technologies. A general advantage of distributed systems is their ability to tolerate service failures better than monoliths since one service crashing may not affect the rest. In our case, for example, if the Subscription Service fails, the Message Service continues as if nothing happened, and only the features of creating, modifying, and

fetching subscriptions are disabled. This is the outcome of SRP and loose coupling between services.

Considering the design, implementation, and results, we are delighted with the outcome and project success overall for most parts. We succeeded to build a system, which actively monitors changes in another system and generates a stream of events based on those changes. We also created an API to interact with the system and control the stream by creating subscriptions. The outcome is to provide employees with valuable information about what is happening in their company, which improves the work efficiency by eliminating routine tasks and keeping them updated. There are numerous of applications utilizing external APIs to provide information to users. However, the Watcher does not use any external APIs, instead it tracks changes in an external database creating a stream of events and is able to deliver information in near real-time. In this way, users do not need to specifically ask for the information by updating their applications, but the information is delivered to them immediately as it appears.

One of the goals we set was to find a general-purpose architecture that can be applied to many different use cases. At this point, it is difficult to evaluate how well we managed to achieve the goal. Nonetheless, the PDM system takes advantage of several databases, which also contain data that the Watcher should be able to access. Therefore, we must add other connectors to the Watcher and implement more logic into the Data Processor. This will also require minor changes in other services as well, but the great thing is that the architecture is done and we can basically just add more incoming data streams to the system.

6.4. Future

The project is completed, but we left little room for enhancements. In the future, we must create automatic tests for each service separately as well as system-wide integration tests. Also, configurations we have defined during the project are only for development, and thus they must be changed before going to production. The same goes for installation scripts, which only work for the development environment.

We have vaguely discussed some client application a couple of times, which is used to manage subscriptions and receive change events as push notifications. Well, the client application will be built in the future, and it is not a separate application like the LSS client but integrated into the PDM system.

Above we presented what to be done before officially launching the Watcher. However, we have some ideas for years to come, such as fully automated delivery, more features, more tests, and orchestration with Docker Swarm. Also, we need to consider how to maintain the Watcher

and what it requires from the administrator or developers. Fortunately, the current understanding is that the system is able to maintain itself, requiring no human interference. This, of course, applies only until the next bug is found or we decide to add more features, which are both inevitable at some point.

7. CONCLUSION

We designed and implemented a real-time watcher system, the Watcher, for tracking changes in another system, PDM, based on user input. The architecture is event-driven in most parts and emphasizes scalability, maintainability, and fault tolerance. Even though we started the implementation from scratch, a legacy version of the Watcher exists, which will be replaced by the new one in the future.

Functional requirements 3.2.1 that were created over the years set the base for the project, which were applied as well as possible. Technical requirements 3.2.2, however, comprise widely known features of modern web applications and were carried out with state-of-the-art technologies.

In the beginning, even before the design, we did a lot of research regarding event-driven architectures and all technologies required to complete this project. The exhaustive literature review contains topics we found the most suitable after carefully evaluating technologies, patterns, and principles singly and together, resulting in a system powered by Spring Boot and Kafka.

The Watcher takes in user-defined subscription objects, which will determine the nature of changes that will be delivered to the users. These users are allowed to define whether the information is delivered by email or as a push event. Currently, the system supports several filters, and more will be released.

The purpose of modernization is to give a better experience overall for customers by providing more powerful tools for running their businesses and thus adding value to their companies as well as ours. The results and evaluation show that we were able to build such a system that has great potential to succeed.

8. REFERENCES

[1] Peltonen H., Martio A. & Sulonen R. (2002) PDM Tuotetiedon hallinta. Edita Publishing Oy, IT Press.

[2] Raja CSP R. & Ludovic D. (2018) Building RESTful Web Services with Spring 5 : Leverage the Power of Spring 5.0, Java SE 9, and Spring Boot 2.0, 2nd Edition., vol. Second edition. Packt Publishing. URL: http://pc124152.oulu.fi:8080/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1699228&site=ehost-live&scope=site.

[3] Fielding R.T. (2000) Architectural styles and the design of network-based software architectures. Publication, University of California, Irvine. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[4] Richardson maturity model. https://martinfowler.com/articles/richardsonMaturityModel.html. Accessed: 2022-09-15.

[5] Ghomi E.J., Rahmani A.M. & Qader N.N. (2017) Load-balancing algorithms in cloud computing: A survey. Journal of Network and Computer Applications 88, pp. 50–71.

[6] Docker overview. https://docs.docker.com/get-started/overview/. Accessed: 2022-09-16.

[7] Docker registry. https://docs.docker.com/registry. Accessed: 2022-09-16.

[8] Merkel D. et al. (2014) Docker: lightweight linux containers for consistent development and deployment. Linux j 239, p. 2.

[9] Boettiger C. (2015) An introduction to docker for reproducible research. SIGOPS Oper. Syst. Rev. 49, p. 71–79. URL: https://doi.org/10.1145/2723872.2723882.

[10] Moraila G., Shankaran A., Shi Z. & Warren A.M. (2014) Measuring reproducibility in computer systems research. PLoS Comput Biol 9.

[11] Manage data in docker. https://docs.docker.com/storage/. Accessed: 2022-09-16.

[12] Dockerfile reference. https://docs.docker.com/engine/reference/builder/. Accessed: 2022-09-16.

[13] About storage drivers. https://docs.docker.com/storage/storagedriver/. Accessed: 2022-09-16.

[14] Docker storage drivers. https://docs.docker.com/storage/storagedriver/select-storage-driver/. Accessed: 2022-09-16.

[15] Share the application. https://docs.docker.com/get-started/04_sharing_app/. Accessed: 2022-09-16.

[16] About the open container initiative. https://opencontainers.org/about/overview/. Accessed: 2022-09-16.

[17] Overview. https://docs.docker.com/compose/. Accessed: 2022-09-16.

[18] Padallan J.O. (2019) Internet 'I&' Distributed Systems. Arcler Press. URL: http://pc124152.oulu.fi:8080/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2013885&site=ehost-live&scope=site.

[19] Kalske M., Mäkitalo N. & Mikkonen T. (2017) Challenges when moving from monolith to microservice architecture. In: International Conference on Web Engineering, Springer, pp. 32–47.

[20] Independent systems architecure. https://isa-principles.org/. Accessed: 2022-09-16.

[21] Taibi D., Lenarduzzi V., Pahl C. & Janes A. (2017) Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages. In: Proceedings of the XP2017 Scientific Workshops, XP '17, Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/3120459.3120483.

[22] Engel Y. & Etzion O. (2011) Towards proactive event-driven computing. In: Proceedings of the 5th ACM international conference on Distributed event-based system, pp. 125–136.

[23] Michelson B.M. (2006) Event-driven architecture overview. Patricia Seybold Group 2, pp. 10–1571.

[24] Kabakus A.T. & Kara R. (2017) A performance evaluation of in-memory databases. Journal of King Saud University - Computer and Information Sciences 29, pp. 520–525. URL: https://www.sciencedirect.com/science/article/pii/S1319157816300453.

[25] Boicea A., Radulescu F. & Agapin L.I. (2012) Mongodb vs oracle – database comparison. In: 2012 Third International Conference on Emerging Intelligent Data and Web Technologies, pp. 330–335.

[26] What is a document database? https://www.mongodb.com/document-databases. Accessed: 2022-10-26.

[27] Mongodb and oracle compared. https://www.mongodb.com/compare/mongodb-oracle. Accessed: 2022-10-26.

[28] What is rocksdb? http://rocksdb.org/docs/support/faq.html. Accessed: 2022-10-27.

[29] How to tune rocksdb for your kafka streams application. https://www.confluent.io/blog/how-to-tune-rocksdb-kafka-streams-state-stores-performance/. Accessed: 2022-10-27.

[30] Narkhede N., Shapira G. & Palino T. (2017) Kafka: the definitive guide: real-time data and stream processing at scale. " O'Reilly Media, Inc.".

[31] When not to use apache kafka? https://www.kai-waehner.de/blog/2022/01/04/when-not-to-use-apache-kafka/. Accessed: 2022-10-27.

[32] Schema registry overview. https://docs.confluent.io/platform/current/schema-registry/index.html#sr-overview. Accessed: 2022-10-27.

[33] Apache kafka needs no keeper: Removing the apache zookeeper dependency. https://www.confluent.io/blog/removing-zookeeper-dependency-in-kafka/. Accessed: 2022-10-27.

[34] Khadka R., Batlajery B.V., Saeidi A.M., Jansen S. & Hage J. (2014) How do professionals perceive legacy systems and software modernization? In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Association for Computing Machinery, New York, NY, USA, p. 36–47. URL: https://doi.org/10.1145/2568225.2568318.

[35] Bisbal J., Lawless D., Wu B. & Grimson J. (1999) Legacy information systems: issues and directions. IEEE Software 16, pp. 103–111.

[36] Cqrs. https://martinfowler.com/bliki/CQRS.html. Accessed: 2022-09-12.

[37] Young G. (2010) Cqrs documents. cqrs. files. wordpress. com .

[38] Jvm ecosystem report 2021. https://res.cloudinary.com/snyk/image/upload/v1623860216/reports/jvm-ecosystem-report-2021.pdf. Accessed: 2022-10-30.

[39] Debezium connector for oracle. https://debezium.io/documentation/reference/stable/connectors/oracle.html. Accessed: 2022-09-13.

[40] Welcome to apache maven. https://maven.apache.org/. Accessed: 2022-10-30.

[41] Jvm ecosystem report 2020. https://snyk.io/wp-content/uploads/jvm_2020.pdf. Accessed: 2022-12-01.

[42] Developing with spring boot. https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.structuring-your-code. Accessed: 2022-11-14.

[43] Code conventions for the java tm programming language. https://www.oracle.com/java/technologies/javase/codeconventions-contents.html. Accessed: 2022-11-14.

[44] Error handling. https://developer.confluent.io/learn-kafka/kafka-streams/error-handling/. Accessed: 2022-11-18.

[45] Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Standard.

[46] Why avro for kafka data? https://www.confluent.io/blog/avro-kafka-data/. Accessed: 2022-12-09.

9. APPENDICES

Appendix 1 Architecture

9.1. Architecture

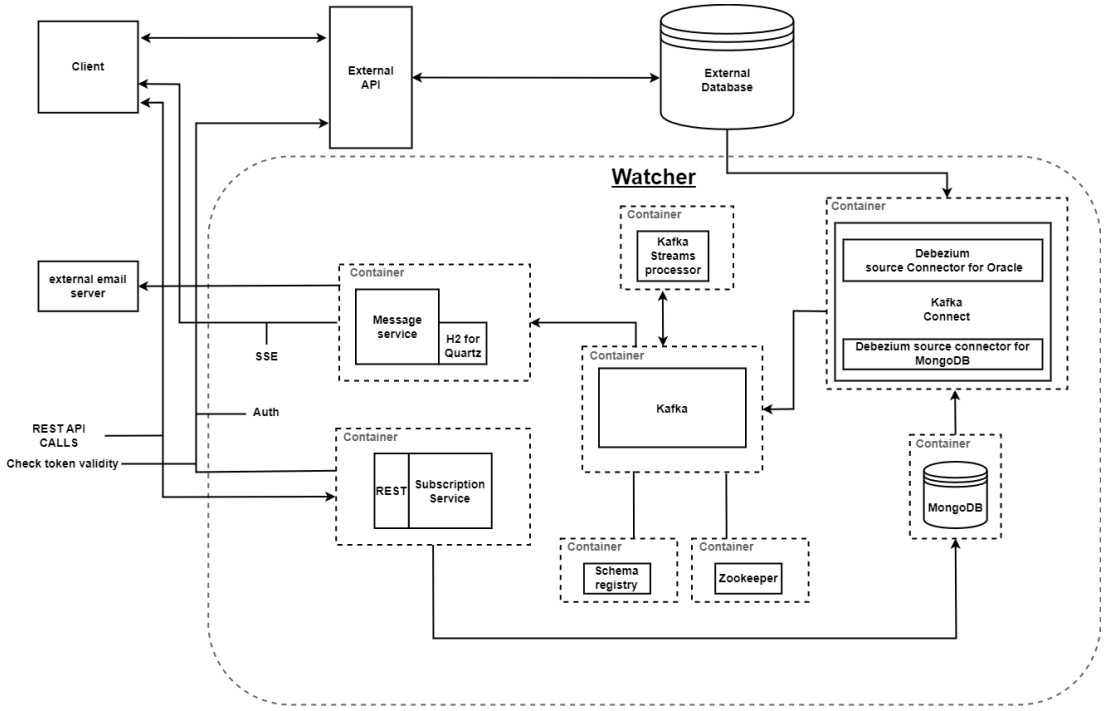

Figure 13. Detailed description of the system architecture