Tampere University

Valtteri Viikari

INTEGRATING A MONOLITHIC WEB SERVICE INTO A MICROSERVICE-BASED CLOUD PLATFORM

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Assistant Professor (tenure track) Outi Sievi-Korte
Associate Professor (tenure track) Davide Taibi
April 2022

ABSTRACT

Valtteri Viikari: Integrating a Monolithic Web Service Into a Microservice-Based Cloud Platform
Master of Science Thesis
Tampere University
Master's Programme in Information Technology
April 2022

Integrating an uneditable monolithic web service into a microservice-based cloud platform is not a trivial task. Even if only a few features of the monolith are desired in the microservice platform, the monolith has to be integrated as a whole. The monolith may have undesirable features, such as user authentication, that the usage of the monolith requires. When integrating such a monolithic web service workarounds have to be developed to avoid the problematic features of the monolith.

The primary research goal of this thesis was to find out how monolithic web services can be integrated into microservice-based cloud platforms without modifying the monolith. The research question was explored by first developing and then evaluating a prototype implementation of integrating Open Cloud (an uneditable monolithic web service) into Vertex Sync (an existing microservice-based cloud platform). The case also gave rise to two more research questions: Is Open Cloud suitable for existing microservice-based cloud platforms and whether Open Cloud is suitable for Vertex Sync.

Open Cloud was deemed suitable for existing microservice-based cloud platforms as the implementation fit the nature of microservices well and the unsolvable issues found during development did not cause enough issues to make Open Cloud unsuitable. Only two issues were found that had a moderate impact on the suitability of Open Cloud by increasing the resource consumption and therefore the expenses of running the system.

Open Cloud was also deemed preliminarily suitable for Vertex Sync as the implementation was judged to be suitable for microservice-based cloud platforms and Open Cloud's performance was judged to be adequate enough. The expenses of running the system require further analysis before Open Cloud can be judged to be suitable for Vertex Sync for sure.

The implemented method of having an intermediary microservice responsible for all communication between Open Cloud (the monolith) and the other microservices was deemed a simple and effective solution to the question of how monolithic web services can be integrated into microservice-based cloud platforms. The method was evaluated to be applicable to most monolithic web services and microservice-based cloud platforms and it can be used to solve most issues related to the integration. However, additional microservices should be considered depending on the implementation environment.

Keywords: Microservices, Monolith, Open Cloud, Technology Adoption

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Valtteri Viikari: Monoliittisen verkkopalvelimen integrointi mikropalveluihin pohjautuvaan pilvialustaan
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Huhtikuu 2022

Monoliittisen verkkopalvelimen integroiminen mikropalveluihin pohjautuvaan pilvialustaan ei ole yksinkertainen prosessi. Vaikka monoliitista tarvittaisiin vain muutama ominaisuus mikropalvelualustalle, monoliitti pitää integroida kokonaisuudessaan. Monoliitilla saattaa olla ei-toivottuja ominaisuuksia, kuten autentikointi, joita monoliitin käyttäminen vaatii. Integroitaessa monoliittista verkkopalvelinta joudutaan keksimään ratkaisuja, joilla ei-toivotut ominaisuudet voidaan välttää.

Työn ensisijainen tutkimuskysymys on miten monoliittinen verkkopalvelin voidaan integroida mikropalveluihin pohjautuvalle pilvialustalle muokkaamatta monoliittia. Tutkimuskysymystä tutkittiin ensin kehittämällä ja sen jälkeen evaluoimalla prototyyppiä, jolla Open Cloud (monoliittinen verkkopalvelin) voidaan integroida Vertex Synciin (mikropalveluihin pohjautuvaan pilvialustaan). Tapauksesta nousi myös kaksi muuta tutkimuskysymystä: Onko Open Cloud soveltuva käytettäväksi olemassa olevissa mikropalveluihin pohjautuvissa pilvialustoissa ja onko Open Cloud soveltuva käytettäväksi Vertex Syncissä.

Open Cloud todettiin soveltuvan käytettäväksi olemassa olevissa mikropalveluihin pohjautuvissa pilvialustoissa, koska luotu prototyyppi sopi mikropalveluarkkitehtuuriin hyvin ja prototyypin kehityksen aikana ei löydetty tarpeeksi merkittäviä ongelmia, jotka voisivat tehdä Open Cloudista epäsoveltuvan. Kehityksen aikana löydettiin kaksi ongelmaa, jotka vaikuttivat vain vähän Open Cloudin soveltuvuuteen. Ongelmat nostivat vain systeemin resurssienkulutusta ja siten systeemin ajonaikaisia kuluja.

Open Cloud todettiin myös alustavasti soveltuvan Vertex Synciin, koska prototyyppi todettiin soveltuvaksi mikropalveluihin pohjautuville pilvialustoille ja Open Cloudin suorituskyky todettiin riittäväksi. Ajonaikaiset kulut vaativat kuitenkin lisää analyysia ennen kuin Open Cloud voidaan todeta varmasti soveltuvaksi Vertex Synciin.

Prototyypissä käytettiin välittäjänä toimivaa mikropalvelua, joka on vastuussa kaikesta kommunikaatiosta Open Cloudin ja muiden mikropalvelujen välillä. Tapa todettiin yksinkertaiseksi ja toimivaksi ratkaisuksi kysymykselle miten monoliittinen verkkopalvelin voidaan integroida mikropalveluihin pohjautuvaan pilvialustaan. Käytetty tapa evaluoitiin soveltuvaksi lähes kaikkiin tapauksiin, joissa monoliittinen verkkopalvelin yritetään integroida mikropalveluihin pohjautuvaan pilvialustaan. Tapa myös todettiin soveltuvaksi tavaksi ratkaista suurin osa ongelmista, joita voi ilmetä integroinnin aikana. Ylimääräisiä mikropalveluita kannattaa kuitenkin harkita toteutusympäristöstä riippuen.

Avainsanat: Mikropalvelu, Monoliitti, Open Cloud, Teknologian käyttöönotto

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

This thesis was written under the employment of Vertex Systems Oy. I would like to thank my supervisor from Vertex Systems Oy, Juhana Rajala, for his time, support, and advice throughout the development of this thesis. I am also grateful to Timo Tulisalmi, Petri Molkkari, and the rest of the Vertex Sync team for letting me work on this thesis in a supportive, low-stress environment and for giving me freedom in deciding where to go with this thesis.

I also need to thank my supervisor from Tampere University, Outi Sievi-Korte, for her constant feedback and help with the writing process. It has been invaluable in improving this thesis throughout the writing process.

Lastly, I want to thank my family for their constant love and support throughout this thesis and my life in general.

Tampere, 27th April 2022

Valtteri Viikari

CONTENTS

1. Introduction . . . 1
2. Web services . . . 3
    2.1 Monolithic web services . . . 3
        2.1.1 Overview . . . 3
        2.1.2 Advantages & disadvantages . . . 5
    2.2 Microservices . . . 6
        2.2.1 Overview . . . 6
        2.2.2 Advantages & disadvantages . . . 9
    2.3 Single-page applications . . . 11
    2.4 Representational state transfer . . . 12
3. Open Cloud . . . 15
    3.1 Overview . . . 15
    3.2 Open Design Alliance . . . 16
    3.3 Front-end components . . . 17
    3.4 Back-end components . . . 17
    3.5 Integrating Open Cloud to a microservice-based cloud platform . . . 18
4. Research Process . . . 20
    4.1 Thesis background . . . 20
    4.2 Research questions . . . 21
    4.3 Research methodology . . . 22
5. Implementation . . . 25
    5.1 Overview of the implementation environment . . . 25
    5.2 Major design decisions . . . 26
    5.3 Implementation details . . . 28
        5.3.1 General overview of the implementation . . . 28
        5.3.2 Overview of the new visualizer service . . . 29
        5.3.3 Other changes within the system . . . 30
    5.4 Future improvements . . . 31
        5.4.1 Improving usability via notifications . . . 31
        5.4.2 Smarter communication methods . . . 31
        5.4.3 Refactoring the implementation . . . 32
6. Observations on the implementation . . . 33
    6.1 Overview . . . 33

6.2 Significant issues found during development 34
6.2.1 User authentication 34
6.2.2 Uploading files to Open Cloud 36
6.2.3 File storage 37
6.2.4 Client.js related issues 38
6.3 Impact of the issues on the suitability of Open Cloud 40
6.4 The implementation's fit for microservices 40
7. Discussion 42
7.1 RQ2: Is Open Cloud suitable for existing microservice-based cloud platforms 42
7.2 RQ3: Is Open Cloud suitable for Vertex Sync 43
7.3 RQ1: How to integrate a monolithic web service into a microservice-based cloud platform without modifying the monolith 44
7.4 Evaluating the suitability of a monolithic web service for a microservice-based cloud platform 46
8. Conclusion 47
References 49

LIST OF FIGURES

2.1 Example of a strictly monolithic web service 4
2.2 Example of the same monolithic web service with external components . . 4
2.3 Example of the same web service implemented using microservices. . . . 7

3.1 A High-level overview of the architecture of Open Cloud based on [34]. . . 18

5.1 High-level architectural overview of Vertex Sync's early development version. 25
5.2 High-level architectural overview of the new implementation. 29

6.1 Sequence diagram of tenant creation. 35
6.2 Sequence diagram of a typical request to Open Cloud. 35
6.3 Sequence diagram of uploading files to Open Cloud via the visualizer service. 37

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application programming interface
BCF	BIM Collaboration Format
BIM	Building information modeling
CAD	Computer-aided design
CRUD	Create, read, update and delete
DOM	Document Object Model
DWG	A proprietary CAD file format
glTF	GL Transmission Format
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IFC	Industry Foundation Classes
JSON	JavaScript Object Notation
ODA	Open Design Alliance
PDF	Portable Document Format
REST	Representational state transfer
SDK	Software development kit
SPA	Single-page application
UI	User interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VSF	Visualize Stream File
VSFX	An extension of VSF
XML	Extensible Markup Language

1. INTRODUCTION

The subject of how, when, and why a monolithic web service should be integrated into a microservice-based cloud platform is a well-researched subject. Several different methods have been developed to ease the transition from a monolith to microservices [1, 2], frameworks have been developed to assess the value of migrating from monoliths to microservices [3, 4], and Sam Newman has even written a book on the subject [5]. All of the sources however assume that the monolith can be edited and therefore decomposed. What if the monolith has to be integrated into an existing microservice-based cloud platform without modifying the monolith?

Open Design Alliance's Open Cloud is a product aimed at helping companies make a cloud product in the field of computer-aided design (CAD). It offers features for viewing various CAD files through a browser interface, converting CAD files into various other formats, and much more. [6] Enabling the features of Open Cloud requires the use of its deployable web server, so integrating it into an existing microservice-based cloud platform requires integrating the server as well. The server is however highly monolithic. It can barely be configured and it requires the usage of its own authentication system and file storage. Integrating Open Cloud into existing microservice-based cloud platforms in a user-friendly manner is therefore a complex task. Open Cloud might also simply be unsuitable to be used in such systems because of its requirements.

The subject of how to integrate a monolithic web service into microservice-based cloud platforms without modifying the monolith is highly prevalent when trying to integrate Open Cloud into a microservice-based system. There does not seem to exist any scientific research on the subject and the only relevant idea related to it is the mention of the decorating collaborator pattern in Newman's book [5]. It is however only very briefly introduced and no research exists on the pattern.

This thesis' goal and primary research question (RQ1) is to find out how can monolithic web services be integrated into existing microservice-based cloud platforms without modifying the monolith? The research question is explored using a case study of integrating Open Cloud into Vertex Sync, a microservice-based cloud platform in development by Vertex Systems Oy. The case also gives rise to two additional research questions. The more generic research question, RQ2, is whether Open Cloud is suitable to be used in

existing microservice-based cloud platforms. In essence, it is about whether Open Cloud can be integrated into an existing microservice-based cloud platform in a user-friendly manner, where the troublesome requirements of Open Cloud can be worked around without causing issues for the entire system. The third research question, RQ3, is simply whether Open Cloud is suitable for Vertex Sync.

Although this thesis' original purpose was to explore the suitability of Open Cloud for Vertex Sync, RQ1 was elevated to be the primary research question and goal of this thesis as it can provide valuable knowledge and help about the subject to the general public. Currently, no applicable research, methods or even general guidelines exists on the subject. During the development of the prototype, the method used to integrate Open Cloud had to be developed from scratch without receiving any help on the subject from relevant literature. With the existence of this thesis future developers of similar systems can receive at least some guidance on how the system should be implemented. This thesis can also serve as a basis for future research to further explore the subject.

The research questions are answered with the help of design science research using a case study for the evaluation. First, a prototype implementation of integrating Open Cloud into Vertex Sync is made which can then be evaluated to answer the research questions. RQ2 is answered by simply evaluating the implementation while disregarding Open Cloud's quality as a product. Then RQ3 can be answered based on RQ2's results and the evaluation of Open Cloud's quality as a product. The evaluation of the quality is however done privately within Vertex Sync. Finally, RQ1 is answered based on the results of RQ2 and by analyzing the method used to integrate Open Cloud into Vertex Sync.

This thesis is presented in eight chapters. After this introductory chapter monolithic web services, microservices, single-page applications, and representational state transfer are detailed to introduce the relevant concepts of this thesis and the integration of Open Cloud into Vertex Sync. The third chapter introduces Open Cloud and the issues related to integrating it into microservice-based cloud platforms. The fourth chapter explains the background for this thesis and further details the research questions and the research process. The fifth chapter gives an overview of the implemented solution for integrating Open Cloud into Vertex Sync. The sixth chapter details all the individual observations that were obtained from developing the implementation. In the seventh chapter, conclusions are drawn from the observations of the sixth chapter to answer the research questions. Finally, the eighth chapter concludes this thesis by briefly overviewing the results and discussing their importance.

2. WEB SERVICES

To be able to evaluate how monolithic web services can be integrated into microservice-based cloud platforms, it is clearly necessary to know what monolithic web services and microservices are, and how they differ from each other. Also in the case of Open Cloud, knowledge about single-page applications and the REST (Representational State Transfer) architectural style is necessary to be able to evaluate the suitability of Open Cloud for microservice-based cloud platforms.

2.1 Monolithic web services

When integrating monolithic web services into microservice-based cloud platforms, it is necessary to know what monolithic web services are, what are their main characteristics and what effects they have on the system. In addition, the advantages and disadvantages of monolithic web services are compared to give a more complete overview of the effects of monolithic web services and why they might be unsuitable to be directly used in microservice-based cloud platforms.

2.1.1 Overview

There does not exist any strict definition of what a monolithic web service is. Monolithic web services are often regarded as a single deployable unit that is responsible for handling everything necessary to provide its business value without relying on any other services [7, 8]. However, sometimes strictly non-monolithic web services are considered to be monolithic by their nature. For example, Newman [8] regards all systems that have to be deployed together to be monolithic and Wolff [9] speaks about deployment monoliths.

Monolithic web services typically consist of one large codebase where all the components of the system are woven together into one large executable application. The application typically has minimal communication with external services and all the critical logic happens inside the service. Monolithic web services can be likened to traditional computer software applications. They typically have all their logic inside the executable and they only communicate with rudimentary components such as IO devices, hard drives, etc. Figure 2.1 shows an imaginary example of a monolithic web service.

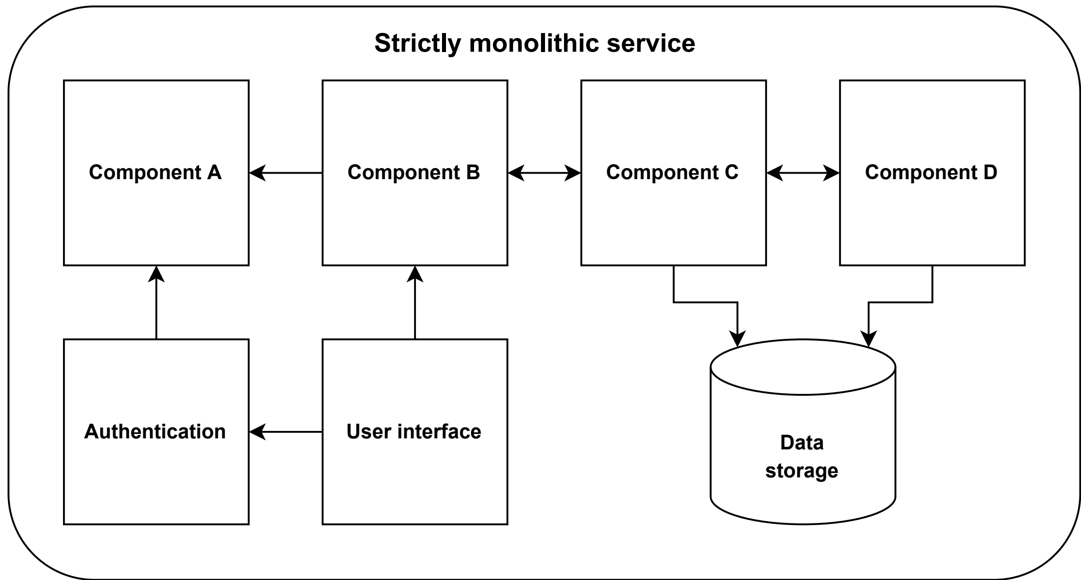

Figure 2.1. *Example of a strictly monolithic web service.*

Monolithic web services are often not pure monoliths as most web services at least have a separate database. They might also have some minor components of the system such as user authentication in another independent service. However, if their main business logic is still in only one service they can be regarded as monolithic. Figure 2.2 shows an example of a strictly non-monolithic web service that would be commonly regarded as a monolith. Also, some systems that were developed following the microservice principle might be monolithic in nature if proper care was not put into service independency and abstraction. Newman refers to such systems as distributed monoliths. [8]

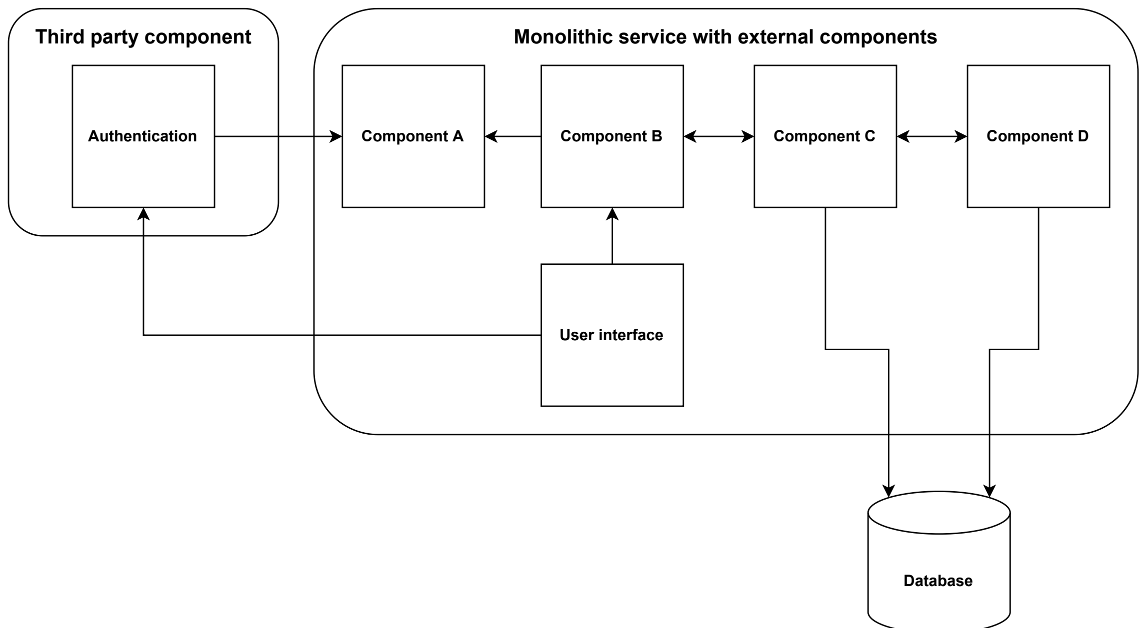

Figure 2.2. *Example of the same monolithic web service with external components.*

Even though monolithic architectures or services are sometimes regarded as a bad sign of a legacy system or in general a badly designed system with mostly downsides [10], they still offer a wide range of advantages and they should always be considered as a valid option to microservices [8]. Some sources even state that monolithic architectures should be the default choice for new applications and the architects should carefully consider why another architecture such as microservices should be utilized [8, 11].

2.1.2 Advantages & disadvantages

The advantages of monolithic web services are fairly well known and universal. Some of the well known significant advantages of (non-distributed) monolithic web services include the following [7, 8, 12]:

Simple deployment Monoliths typically consists of a single or at least very few independent parts so the deployment process is typically quite simple. For the so-called distributed monoliths, the situation might be very different.

Easier testing Testing a monolithic web service is much easier compared to microservices as all parts of the system are guaranteed to be working together. No extra effort has to be put into orchestrating multiple services or mocking the behavior of other services.

Code reuse Code reuse is trivial in monolithic systems. All the source code is available to all components as they all share the source code.

Easier to develop As the monolithic architecture can be thought of as the default architecture and the developers do not need to know about any other services or their communication, getting into development can be much easier. For old, complex monoliths the situation may be very different.

While the advantages of monolithic web services are fairly universal the disadvantages are much more situational. So whenever the architecture is being considered for a new service, one should consider which disadvantages are actually relevant to the system. Some of the well-known disadvantages include the following [7, 8, 12]:

Can be hard to understand As all the logic for a monolith is contained within a single codebase, gaining a sufficient understanding of the system may be very hard. Some seemingly meaningless parts of the source code may unexpectedly affect the entire system.

Intertwined logic Minor changes to the source code in a monolithic system may require making a lot of changes to multiple components. It may also have unexpected

consequences in other parts of the system.

Bad scalability If one part of a monolithic system is under heavy load the only way to scale it is to scale the entire system. This causes all the other parts not under heavy load to be scaled too easily causing extra expenses.

Tight coupling of components As a monolith is deployed as one large mass it requires all the components of the system to be working. Issues in one component cause problems for the entire application and a single major problem in one part of the system can bring the entire system down.

Locked into selected technologies Monolithic systems are typically locked into the technologies selected at the start of development. Changing the underlying technologies would require extensive reworks all over the system.

One additional major disadvantage of monolithic web services that seems to be rarely listed anywhere is that monolithic systems can be hard to integrate into other systems. If the service does not have a lot of configuration options, or methods to modify or extend the functionality, monolithic services have to be integrated as-is. In more modular systems, such as microservice-based systems, the functionality could be much more easily modified. Custom logic and components could be added between parts of the system and minor components could even be replaced or remade. This issue becomes very apparent in this thesis as integrating Open Cloud to Vertex Sync requires many workarounds for parts of Open Cloud's logic.

2.2 Microservices

To be able to evaluate a monolithic web service's suitability for a microservice-based cloud platform, it is also clearly necessary to know what microservices are, what characteristics they typically have, and also the effects they cause on the complete system. A deeper understanding of microservices' characteristics and advantages is also necessary as the suitability of monolithic web services for microservice-based cloud platforms is highly dependent on them.

2.2.1 Overview

Microservices, as the name suggests, differ wildly from monolithic services by their core concept. In microservice-based architectures, the business domain is separated into loosely coupled, fine-grained services that are only interested in completing their own responsibility. Microservices are used in great effect in modern web platforms, which can be seen through the success of services such as Microsoft Azure, Amazon Web Services,

and Google Cloud. For example, Netflix utilizes over a thousand different microservices to serve its video streaming platform [13].

Typically each microservice is an independent service modeled around a particular area of the business domain. They have a well-defined application programming interface (API) that they try to satisfy. The implementation and technology details on how each microservice satisfies its API are irrelevant and should in fact be hidden from the users of the API. Communication between different microservices happens through simple, lightweight protocols such as HTTP (HyperText Transfer Protocol) or message buses. [8]

A simple microservice variant of the service from figure 2.2 is depicted in figure 2.3. Both the monolithic and the microservice variants provide the exact same functionality. In the microservice variant, the implementation is split into multiple services, and each service could be developed independently from the others even with completely different technologies. Service A could be a dedicated machine running a Java server and service B a simple Node server running inside a container.

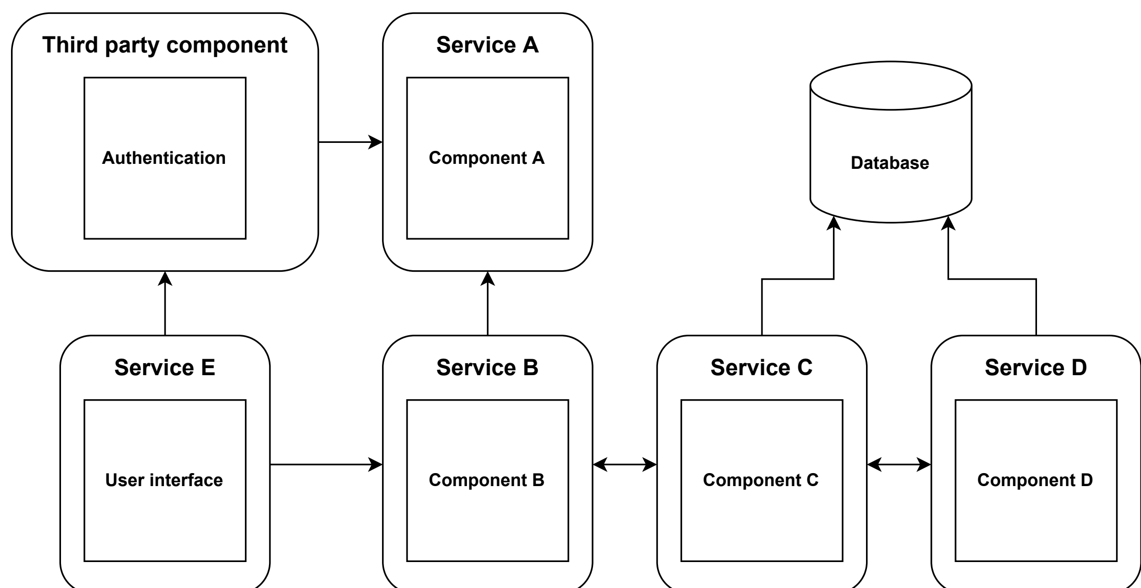

Figure 2.3. *Example of the same web service implemented using microservices.*

There does not exist a strict definition for microservices and there exist a lot of different views of what microservices are, what they contain, and what are not microservices. One of the most common definitions of what microservices are comes from Amundsen et al. [14]. They distill microservices into 7 core characteristics:

- **Small**
- **Messaging enabled**
- **Bounded by contexts**

- **Autonomously developed**
- **Independently deployable**
- **Decentralized**
- **Built and released with automated processes**

Another alternative definition of microservices key characteristics comes from Fowler & Lewis. They define the core concepts of microservices to be [15]:

- **Componentization via services**
- **Organized around business capabilities**
- **Products not projects**
- **Smart endpoints and dumb pipes**
- **Decentralized governance**
- **Decentralized data management**
- **Infrastructure automation**
- **Design for failure**
- **Evolutionary design**

There also exists myriad other equally valid definitions and lists of microservice characteristics (like Sam Newman's [8]). The variance of the definition of microservice key characteristics shows how loosely defined the concept of microservices is in practice. Some of the most common characteristics are further detailed below [8, 14, 15]:

Small size As the microservices term itself suggests services in the architecture are usually of small size. There do not exist any limitations to how big a service can be or how small it should be kept, but microservices should be kept at a level where the service can be easily understood. If a service starts getting too big to not be easily understandable, it might be worth re-evaluating the responsibilities of the service or splitting the service into multiple smaller services.

Flexibility Flexibility is one of the core characteristics and benefits of microservices. Nearly all the other characteristics bring, but also require, a huge amount of flexibility. Flexibility does not come for free though as increasing the flexibility often also increases the costs of microservices [8].

Independent deployability Microservices should be possible to be independently deployed without individual services impacting others. It enables making changes to a live system without having to interrupt the entire system. Independent deployability requires and also results in loose coupling between microservices.

Modeled around a business domain Service boundaries in microservices are defined by the business domains. This naturally creates an easily understandable division of responsibilities. This can be achieved by following techniques such as domain-driven design during development [8].

Autonomously developed & Alignment of architecture and organization In companies developing microservices developer responsibilities are often determined by the services they work on rather than the traditional way of dividing work into front-end, back-end, database, etc. This way different services can easily utilize wildly varying technologies as only the people responsible for the service need to be knowledgeable about them.

Decentralized Microservices can be easily distributed into different regions and platforms without needing any sort of central orchestrator or communicator. This brings in the benefits commonly associated with distributed systems.

2.2.2 Advantages & disadvantages

The core advantages of microservices can be derived from their core characteristics. Also, all the advantages of distributed systems (scalability, modularity, resiliency) [16] are applicable to microservices. Microservices tend to benefit from the advantages of distributed systems to a greater degree than normal because of their well-defined service boundaries and their concept of information hiding [8]. The advantages of microservices are fairly synonymous with the disadvantages of monolithic web services. Some of the most significant advantages of microservices include the following [7, 8, 9, 17]:

Scalability One of the most recognized benefits of microservices is their ability to easily scale. Unlike in monolithic systems where scaling a particular area of the system would require scaling the entire application, in microservices scaling can be applied to individual services. Scalability does not come by default though and care must be put into enabling scalability. Scalable services can be automatically scaled using technologies such as load balancers to achieve even better results.

Technology agnostic As microservices only communicate with each other using well-defined protocols, like HTTP, the technologies required for implementing an individual microservice can be chosen freely. Even things such as the databases that each service utilizes can be chosen freely. One service could utilize a traditional relational database and another a graph-based one.

Fault tolerancy & service resiliency Microservices only communicate with each other through interfaces, so a small fault or even a missing service would not bring the entire

system down. Only some of the features would become unavailable. Afterwards, the faulty service can be separately fixed and deployed back into the system without significantly impacting the other services' operation.

Reusability Ideally a microservice is not dependent on any other service and it is responsible for only a small domain. Such services can easily be integrated into completely different systems with only minor adjustments.

Individual services easy to understand As a single service is only responsible for one small business domain, understanding the service becomes easy. This enables developers to more easily start developing a specific service.

As with any architectural pattern, microservices also come with several disadvantages. Many of them once again can be traced back to distributed systems. The disadvantages of microservices are quite synonymous with the advantages of monolithic web services. Some of the most significant disadvantages include the following [8, 9, 12]:

Hard to test While unit tests are simple to do in microservices, integration and system testing are much harder tasks. The tests require code to be executed in multiple different services running in different processes, machines, or even regions. In realistic test environments, the actual communication protocols should be used which would open up the possibility of the tests failing due to external circumstances such as network failures.

Monitoring & reporting In non-distributed services monitoring and reporting are simple; all the data and logs are in a single location. In microservices, the situation is much harder to manage. Reporting may require joining multiple databases together that may even have the data stored in completely different formats. Monitoring also becomes harder as there are significantly more things to monitor.

Security In a monolithic service all the data is contained within the single service with nothing running over the network. In microservices, an endpoint may require the service to call other services behind the scenes making data flow across the network. Such data makes the system vulnerable to man-in-the-middle attacks or general unwanted observation. Therefore microservices require extra focus on security.

Latency Microservices often require communication and data flow between services. A complex operation may require a service to ask for data from multiple different services in a consecutive manner. Depending on the communication methods and distances it may add a lot of latency to the request.

As with any list of advantages and disadvantages, the statements should not be taken at face value. The meaningfulness of each advantage and disadvantage is highly dependent

on the system being evaluated. Also, many of the statements can be situational and many of the disadvantages can be mitigated or even avoided with careful planning.

In most situations, the advantages of microservices highly outweigh their disadvantages. This can be concretely seen from the huge popularity and success of microservices [18].

2.3 Single-page applications

Modern web applications (both monolithic and microservice-based ones) often utilize a single-page application in their front-end. For Open Cloud to be suitable for existing microservice-based cloud platforms at all, it must be well suited to be used in single-page applications. Knowledge about the way single-page applications work is necessary to understand why Open Cloud was integrated the way it was into Vertex Sync.

Traditional web applications have a simple workflow for rendering something to the user. First, the user makes a request to a server that it wants to render something. Then the server responds with an HTML (HyperText Markup Language) document containing the relevant data and the presentation to view the data. Finally, the user's browser does a UI (user interface) refresh and renders the received HTML. If the user wanted to see something else he would typically need to click a hyperlink on the page or navigate manually to another URL (Uniform Resource Locator). Modern web applications typically use a completely different, more sophisticated workflow to achieve a better user experience. [19]

Modern web applications typically implement the single-page application (SPA) model. In this model whenever a web application is first accessed, the browser is only served a minimal HTML document accompanied by a large number of script files. When the HTML page is rendered by the browser, the script files manipulate the DOM (document object model) to render different things based on the user's interactions. Whenever external data is needed for rendering some information, the data is fetched asynchronously from a server typically in JSON (JavaScript Object Notation), XML (Extensible Markup Language), or some other data-oriented format. The data is then parsed by the scripts and the DOM is modified to render the newly retrieved data. [19] This kind of workflow clearly implements the essence of the model; the application consists of only a single page.

Single-page applications are typically created with the help of a front-end framework that already implements the model, such as Angular, React, or Vue [20, 21, 22]. Development in such frameworks is usually done in a component-driven manner. Elements of the web page are declared in components that handle how the elements should behave. In addition to any normal user interaction-related code, they also typically have some framework-specific lifecycle methods that define what happens to the components throughout their lifespan (initialization, destruction, etc.). Components can also refer to

each other to create a hierarchical structure between them. [23, 24, 25]

Single-page applications typically have to load the majority of the application logic and rendering-related scripts upon first accessing the site. This enables SPAs to have a smoother user experience where the user does not need to wait for individual pages to load, but it also brings in the greatest flaw of SPAs: the initial load times of SPAs are much slower [19].

2.4 Representational state transfer

Single-page applications need the ability to exchange specific data with the server instead of just HTML documents. Different architectural styles have been developed for back-end servers to work in such situations. One of the most famous of such architectural styles (and also the one that Open Cloud utilizes) is representational state transfer. Knowledge about REST is also necessary to understand why Open Cloud was integrated the way it was.

Representational state transfer, also known as REST, is an architectural style first introduced by Roy Fielding in the year 2000 [26]. REST only defines how a distributed hypermedia system (a distributed cloud platform) should behave and it does not have any specific rules or constraints on how to implement REST nor does it require any specific technologies to be used. Although REST is commonly associated with HTTP, it does not strictly require the usage of HTTP. The REST architectural style can be defined by six constraints [26, 27, 28]:

1. **Interface uniformity** The core constraint of REST is its uniform interfaces. In REST, interfaces are designed to be uniform so that manipulating different resources can be done through uniform methods. This enables REST applications to be loosely coupled as the implementation details are hidden from the APIs. It does however decrease performance as the data and requests are required to be in a standard format rather than the most efficient one.

   To achieve interface uniformity, REST defines four additional constraints for interfaces: Identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state.

2. **Client-server architecture** The client-server architectural style in essence refers to the separation of systems into requesters (the front-end) and providers (the back-end). The REST architectural style requires the constraints from the client-server architectural style. The constraints are mainly required for the separation of concerns principle. It enables the UI of REST-based applications to be highly portable while allowing the server to be highly scalable. It also enables independent development of the front-end and the back-end.

3. **Statelessness** All communication in REST should happen through stateless methods. Each request must be able to be completed without relying on any session state stored in the server. Statelessness increases the visibility, reliability, and scalability of the server as each request can be handled independently regardless of the server's state. However, it also decreases performance by requiring the client to send repetitive data.
4. **Cacheability** Data in REST must be implicitly or explicitly declarable to be cacheable. Cacheable data increases efficiency, performance, and scalability as some requests can be avoided if the result has already been cached. It may decrease reliability though, as the cached data may differ from the data that the server would have sent.
5. **Layered system** In layered systems a component interacting with another one may only see the component that they are interacting with. This allows placing proxies, load balancers, other services, etc. behind the service being called without needing to modify the request, or without the caller needing to know about it. This increases maintainability and scalability by adding the option of being able to modify requests easily. It may also increase security as security can be extracted to a separate layer from the business logic. However, it may also increase latency as a single request may require many components to execute code.
6. **Code-on-demand** The final constraint for REST is the ability of the server to extend client functionality by sending code to the client in the form of scripts. It may simplify client functionality but it can also decrease visibility. The code-on-demand constraint is optional.

Web services that implement the REST architectural style are called RESTful web services. RESTful services do not have any standard on how they should be implemented (nor does REST in general) so the general practices of RESTful services can be found in a multitude of different sources. Some of the sources have even contradictory rules. For example, Giessler et al. [29] state that verbs should not be used in URI's (Uniform Resource Identifiers) at all, and Masse [30] states that verbs should be used in URIs for controller names. The high-level details about how RESTful web services should be implemented are equal in most sources [29, 30, 31, 32]. The following chapters give an overview of some of the most common practices of RESTful web services based on Richardson et al. practices [31].

In RESTful web services, all the data of the system is abstracted behind resources. A resource is typically either an object that represents something (a person, a document, etc.) or a list of individual resources. Resources and the operations that can be performed on them are made available through well-defined URIs paired with HTTP verbs.

The URI of an item is formed by its resource's name and its status in the hierarchy. The

names and the hierarchy should be easily understandable and predictable to make the service intuitive and easy to use even without reading the documentation. For example, if a system kept track of its users and their preferences, the list of all users would be available through the URI "/users". An individual user's data would be available by specifying the ID of the user within the resource. For example, "/users/1" would be the URI of the user with 1 as their ID. That user's preferences would then be available through the URI "/users/1/preferences".

The operation that the user wants to perform on a resource is determined by the HTTP verb they use on the HTTP request. The most common operations, create, read, update and delete (the CRUD operations), are available through the HTTP verbs POST, GET, PUT (or PATCH) and DELETE respectively. There also exists a wide range of other HTTP verbs for more specialized situations such as OPTIONS or HEAD.

The CRUD-related HTTP verbs do not match CRUD operations perfectly. The GET verb is used to get the content of a resource from the server. The server must not normally modify any data while serving a GET request, but sometimes it is allowed to enable tracking or monitoring of a resource [31]. The update operation of CRUD is often implemented using two verbs, PUT and PATCH. Typically PUT replaces all the data of the specified resource while PATCH only modifies the values present in the request.

Status code	**Description**
200 OK	The request was a success
201 Created	New resource created
204 No Content	The request was a success but no body was returned
400 Bad Request	Invalid request (client error)
401 Unauthorized	The user does not have proper authentication
403 Forbidden	Access forbidden regardless of authorization
404 Not Found	Resource was not found
500 Internal Server Error	The server ran into an error

Table 2.1. *Commonly used HTTP status codes [33]*

The result of a request in RESTful web services can be read through the response's body and HTTP status code. The meaning of the different status codes comes straight from the RFC 2616 HTTP/1.1 standard [33]. The first number of a status code represents the category the result belongs to. Status codes starting with 1 represent informational responses, 2 represent success, 3 represent redirection, 4 represent client errors and 5 represent server errors. Some of the most common status codes are detailed in table 2.1.

3. OPEN CLOUD

This chapter briefly introduces Open Design Alliance, Open Cloud, and its architecture. Also, the challenges of integrating Open Cloud to an existing microservice-based cloud platform are described to give a better understanding as to why Open Cloud was integrated the way it was, and to give background information as to why Open Cloud might not be very suitable for an existing microservice platform.

3.1 Overview

Open Cloud is a project developed by Open Design Alliance (ODA) aimed at "helping companies make a cloud product or service in the field of CAD design, and creation and management of BIM [building information modeling] projects". Unlike most of ODA's products, Open Cloud is not a software development kit (SDK) but a complete web service consisting of multiple different components split between the back-end and the front-end. Open Cloud does utilize many of ODA's SDKs in its implementation. [34]

The core features as listed by ODA are roughly as follows [6, 34]:

- Store and manage CAD and BIM files via REST API
- Store and update user information
- Deploy in any environment without connecting to third-party services
- Run in Microsoft Azure, Amazon AWS, or in private cloud
- Extract geometry and property data from various file formats
- Export files into PDF (Portable Document Format) and other formats
- Visualize geometry in a browser

Open Cloud also has several other features that are not listed anywhere. A lot of them are directly inherited from the different SDKs Open Cloud utilizes. Some of the features are not even described anywhere and they are only visible by browsing Open Cloud's documentation or the release notes for previous versions. Most notably BCF (BIM Collaboration Format, a file format/service intended for issue tracking between different BIM applications) support and model assemblies are not listed as features.

The components of Open Cloud can be divided between the front-end and the back-end components. The front-end's main component is a JavaScript library that utilizes WebAssembly for rendering geometry that the back-end service serves. The back-end's main component is a standalone RESTful .NET web server that is responsible for all communication between the back-end and the front-end.

The server is a black box that is mostly not extendable or configurable. The only configurable parts are basic storage-related settings such as maximum file size and the database name and connection string. The only point of extensibility is custom jobs that can be added for certain endpoints. The endpoints need to be explicitly called for the custom jobs to be executed.

Open Cloud is still being actively developed, so many of the issues and lacking aspects found in this thesis might get fixed over time. Some of the faced issues were even improved during the implementation process of this thesis. Also, many of the components of Open Cloud as of writing this thesis are less than half a year old, so they are still very new and they might undergo large changes.

3.2 Open Design Alliance

Open Design Alliance is a nonprofit organization whose mission is to create interoperability between CAD and BIM products [35]. It was formed in the late 1990s with its first product being OpenDWG, a read/write library for working with DWG files [36]. Since then, ODA has developed a suite of SDKs and other products centered around different CAD and BIM technologies. Their customers include technology giants such as Microsoft and Epic Games, and big contributors in the CAD business like Autodesk and Solibri [37].

Open Design Alliance offers multiple different SDKs and toolkits for working with CAD and BIM products in different product areas. For example, they offer SDKs for viewing CAD models, publishing CAD-related files, and converting different proprietary CAD file formats into different ones. [38] Open Design Alliance has started partnerships with multiple different companies to provide better support for some technologies. Notably, they have established a partnership with BuildingSMART for IFC (Industry Foundation Classes) and BCF support [39].

Open Design Alliance's product overview and documentation for some products (Open Cloud) are freely available online, but accessing the files needed for development require an ODA membership. There is no free level of membership and the highest level of membership gives access to all the products' source codes. Most products also have a free 60-day trial.

3.3 Front-end components

The front-end of Open Cloud consists of two different components, one of which is optional. The core component, VisualizeJs, is a dynamically downloadable JavaScript library for rendering CAD models in the browser. It utilizes WebAssembly in its implementation for better performance. VisualizeJs can render dynamically created geometry, glTF (GL Transmission Format) files, and ODA's own proprietary file formats VSF (Visualize Stream Format) and VSFX (a new extension of VSF). The VisualizeJs library does not communicate with any other components or services. All operations and data have to be explicitly called or fed to the library.

The second front-end component is the optional Client.js library. The library has a few different core features all of which are not necessary for it to be used, but they make working with the Open Cloud server easier. Client.js is quite a new addition to Open Cloud, with its first release being in autumn 2021 [40].

The main feature of Client.js is its ability to integrate VisualizeJs to the Open Cloud server. The process of creating an instance of the viewer to viewing a file in the browser from the server can be done in just a few lines of code. Without Client.js replicating the same process would take closer to a hundred lines of code.

Client.js also has some built-in functions for working with VisualizeJs. For example, it has tools for basic camera functionality (mouse/touch controlled pan, orbit, and zoom), basic features such as plane cutting, measuring, etc. Client.js can also be used to communicate with the Open Cloud server easily.

Open Cloud also includes an example implementation of a working React-based front-end that utilizes all parts of the Open Cloud package. It can be used as a model on how to work with Client.js and it works as a clean showcase of a lot of the features of Open Cloud. It is also available as a free online demo on the ODA website [41].

3.4 Back-end components

The structure of the back-end of Open Cloud is significantly more complex than the front-end. The back-end consists of three distinct components: The server itself, a MongoDB database, and at least one JobRunner. Also, another server is required for serving the front-end. A visual high-level overview of the system can be seen in figure 3.1.

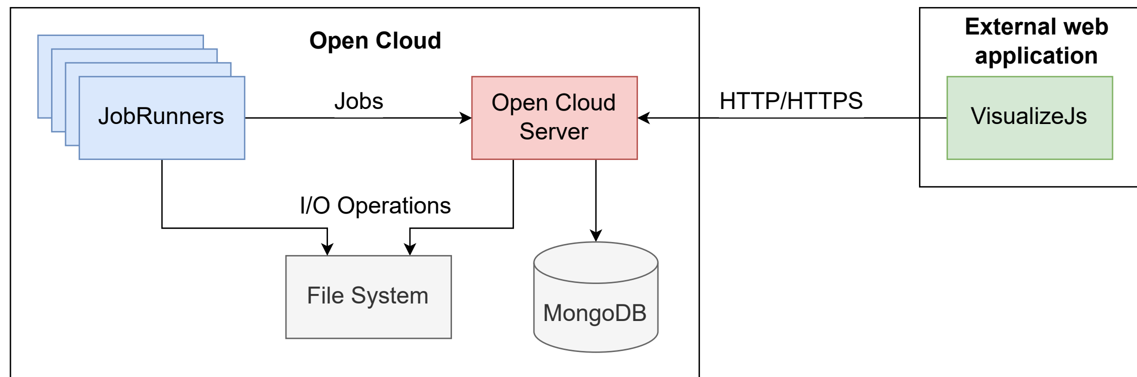

Figure 3.1. *A High-level overview of the architecture of Open Cloud based on [34].*

The server component is fairly lightweight even though it has the most responsibilities. Its main task is user authentication and communication with the other components of the system. The MongoDB database is used for storing miscellaneous data such as user data. The user uploaded files are stored into the file system of the machine running the server (or alternatively to a Microsoft Azure BLOB or an Amazon AWS S3 instance).

The JobRunner component is the component that does all the heavy operations. It is responsible for executing whatever tasks (jobs) that have been registered to the server. File conversions are the most typical jobs that the JobRunners have to execute. Whenever a user uploads a file to the server that they would like to view in the front-end, the JobRunner has to first convert the file into VSF or VSFX format. As the file conversions can be very slow and heavy operations, the jobs the server wants the JobRunners to execute are put into a queue, so the server does not have to wait for the job's execution. Whenever a job is in the queue and a JobRunner is inactive, the inactive JobRunner starts the oldest available job. One server can utilize multiple JobRunners to enable parallel processing of jobs.

3.5 Integrating Open Cloud to a microservice-based cloud platform

As the Open Cloud server is a black box complete with all parts of the system that drive its logic (authentication, routing, file management, etc.), its architecture is highly monolithic. Almost none of the components can be configured or disabled so nearly all parts of the system must be used as-is. This makes Open Cloud not very ideal to be integrated into a microservice-based cloud platform as its authentication system, file management system, etc. all would have to be used. Files will have to possibly be duplicated and there will be two different authentication systems. Integrating it in a user-friendly manner either requires lots of extra work or workarounds for things like authentication.

The easiest way to integrate Open Cloud into a microservice-based cloud platform would

be to build all the microservices around Open Cloud. Open Cloud could be used as the central authentication system, file storage, etc. and all the other microservices call its endpoints to access data, store files, authenticate, etc. If building all the other microservices around Open Cloud is not an option, the other option would be to build workarounds for certain parts of the system. From the perspective of a user, this can be done in a seamless way. However, from the perspective of the developers/hosts, it will cause some issues.

It is possible to use the file converters and the front-end viewer separately from the server. It would require lots of extra work in the front-end as the Client.js library could not be used and all the communication would have to be developed by hand. It would also require at least partially remaking the functionality of the Open Cloud server. All extra functionality of the server like BCF, projects, and viewpoints would be lost without also having to develop them by hand. Any upcoming functionality that would normally be plug-and-play with the server could require lots of extra work to implement. Also on top of all that, maintaining the system would require much more effort.

4. RESEARCH PROCESS

This chapter introduces the background of this thesis, the research methodology, and the process the methodology was implemented with. Also, the research questions are further detailed to give a clearer understanding of the novel contributions of this thesis.

4.1 Thesis background

Vertex Sync is a new cloud product under development by Vertex Systems Oy. The vision of the product is still under development, but the initial vision of the product can be condensed into three points: work anywhere, any time; share documents and data with stakeholders; and team collaboration in large projects (simultaneous design). In essence Vertex Sync is meant to ease the development of projects developed with Vertex Systems' CAD products, by integrating them into a common cloud, to enable working with multiple computers and concurrently with other people. It is also intended to ease sharing of projects between different stakeholders by having all the data (drawings, models, reports, etc.) in one common place easily viewable for all relevant parties.

Vertex Sync is a multi-tenant system, meaning that it is a system where "multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant" [42]. The multi-tenancy aspect of Vertex Sync has a minor effect on the suitability of Open Cloud for Vertex Sync.

One of the core requirements for Vertex Sync is the ability to view different CAD models (especially IFC) online through its browser UI. There do not exist any specific plans on how it should be implemented. One option was to integrate one of Vertex Systems' other products, Vertex Showroom, into Vertex Sync. Vertex Showroom fits the use case well as it is intended for viewing CAD models online, however it requires the use of glTF files along with some custom data received from Vertex Systems' CAD products. It would therefore also require the integration of Vertex Systems' CAD products into Vertex Sync's back-end to convert CAD models into the extended glTF format that Showroom requires.

Vertex Systems has already been utilizing some of ODA's SDKs for years in their CAD products, such as Drawings SDK (formerly known as Teigha). Vertex therefore already

had a subscription to the entire ODA platform so they could utilize Open Cloud without any additional fees. Open Cloud's general overview seemed promising so closer inspection of the product started. The inspection however raised some doubts about whether Open Cloud would be a good fit for Vertex Sync because of its monolithic nature. The doubts were hard to clear by just inspecting the system so Vertex Systems decided to implement a prototype to see whether it would be a good fit or not. The prototyping was done in the form of this thesis.

Vertex Systems did not set any particular requirements for this thesis. The prototype had to be compatible with the existing parts of the system and only minor modifications should be done to the other services (for example no technology changes or switching out the authentication service). The only major effect this had on the requirements was that Open Cloud and its components had to be containerizable with Docker.

4.2 Research questions

This thesis has three research questions. The first research question, RQ1, aims to give generalizable knowledge about this particular case: **How to integrate a monolithic web service into a microservice-based cloud platform without modifying the monolith**? All somewhat relevant research focuses on decomposing the monolith into suitable chunks before integrating it into microservices. In cases like Open Cloud, the monolith has to be integrated in its entirety without being able to modify it at all. Newman briefly introduces the decorating collaborator pattern that can be used in such situations [5], but it has not been scientifically researched. This thesis aims to provide a way monolithic web services could be integrated into microservice-based cloud platforms without modifying the monolith by providing an example case of how the integration can be done.

The second research question, RQ2, aims to answer whether Open Cloud can be utilized in existing microservice-based platforms: **Is Open Cloud suitable for existing microservice-based cloud platforms**? It is important to note that this thesis only answers whether Open Cloud is suitable for *existing* microservice-based platforms or not. In existing microservice-based architectures, Open Cloud has to be built around the other microservices. In new microservice platforms, Open Cloud could be treated as a central service and the other services could be built around it. Open Cloud could therefore be suitable for new microservice-based platforms even if it was not suitable for existing ones.

The third research question, RQ3, is based on Vertex Sync's need for Open Cloud: **Is Open Cloud suitable for Vertex Sync**? It has less importance to the general public, but it is of the utmost importance to the future of Vertex Sync. The answer to this research question is highly dependent on the second one, but it is not completely dependent. Vertex Systems could for example deem it unsuitable for Vertex Sync because of performance issues even if it was suitable for existing microservice-based cloud platforms.

The research questions are answered based on the evaluation of a prototype implementation of integrating Open Cloud into Vertex Sync. First, RQ2 is answered based on analyzing different aspects of the implemented solution while disregarding Open Cloud's quality as a product. Then RQ3 is answered based on the results of RQ2 and Vertex Sync's stakeholders' analysis of the solution's and Open Cloud's quality attributes such as usability, performance, etc. Finally, RQ1 is answered based on the results of RQ2 and by analyzing the method used to integrate Open Cloud to Vertex Sync in the implemented solution.

4.3 Research methodology

Design science research methodology was chosen as the research methodology for this thesis as it fits the nature of the research questions very well. The primary guideline for design science research is that the result produces "a purposeful IT artifact created to address an important organizational problem" that should also be the solution to "heretofore unsolved and important business problems". The result of this thesis does just that. The end result is a working addition to the existing microservice system to solve the business problem of how CAD models should be viewed in Vertex Sync. [43]

The evaluation of the artifact is an essential part of design science research. One of the possible methods that can be used for the evaluation is a case study, which is also the method used in this thesis. [43] Briefly, a case study is the study of a single case for the purpose of understanding a larger class of similar cases [44].

As there does not exist any research on how a monolithic web service should be integrated into a microservice-based cloud platform without modifying the monolith, a case study is an ideal way to explore the question. In this thesis, the integration of Open Cloud into Vertex Sync is used as a case study to answer the general question of how a monolithic web service can be integrated into a microservice-based cloud platform without modifying the monolith.

This case should be quite an ideal one to be used to answer the question as both Open Cloud and Vertex Sync are fairly generalizable to an arbitrary monolithic service and an arbitrary microservice-based cloud platform. Vertex Sync is a traditional microservice-based cloud platform that has no special features, limitations, or attributes that would make it significantly different from the common microservice-based cloud platform. Its only somewhat unique aspect is its multi-tenancy, but it has no impact on the integration. Open Cloud is also a very generic monolith. It has many of the common features one would assume a monolithic web service would have, such as its own user authentication and file storage, and it is not missing any vital features monolithic web services often have.

There exists a variety of processes of how design science research should be implemented. For example, Vaishnavi & Kuechler [45] define a five-step process and Peffers et al. [46] define a process that has an additional sixth demonstration step. In this thesis Peffers et al. process is followed. The steps are as follows:

1. **Identification & motivation of the problem**
2. **Defining the objectives of a solution**
3. **Design & develop a solution**
4. **Demonstrate the developed solution**
5. **Evaluate the developed solution**
6. **Communicate the results of the solution**

The process is also often iterative. After a step is completed, it might be necessary to return to an older step to further refine the solution. In addition, the results of each step should be communicated well. Each step of the process provides valuable knowledge that is used as the output of design science research.

The motivation & identification of the problem (step 1) for this thesis comes mainly from Vertex Systems Oy and the general issues related to Vertex Systems' problems. One of the core requirements for Vertex Sync is the ability to view varying CAD models through its browser interface. Open Cloud provided a great solution for the problem, but it was unclear whether Open Cloud could be used in Vertex Sync or not. This resulted in RQ3, which was further derived into the more general case of Open Cloud's suitability for existing microservice-based cloud platforms (RQ2). There did not exist any relevant literature as to how a monolithic web service like Open Cloud should be integrated into existing microservice-based cloud platforms without modifying the monolith, so RQ1 was added as this thesis could also be an ideal case to provide a generalizable way to solve the issue.

The objectives of the solution (step 2) come straight from the research questions. The objectives are to find out how a monolithic web service can be integrated into a microservice-based cloud platform without modifying the monolith and whether Open Cloud is suitable to be used in existing microservice-based cloud platforms and Vertex Sync.

The design and development (step 3) of the solution were done privately within Vertex Systems. The results of the development and some of the design-related requirements are detailed in chapter 5. The demonstration of the developed solution (step 4) was also done privately within Vertex Systems to the stakeholders of Vertex Sync.

The evaluation step (step 5) is detailed in chapters 6 and 7. As there does not exist any directly applicable research on how to evaluate the suitability of a monolithic web service for a microservice-based cloud platform without modifying the monolith, the evaluation

was based on the developed solution's fit for microservices and the issues found during development. The criteria and the observations related to them are detailed in chapter 6. The results of the criteria are further analyzed to answer RQ2 in chapter 7. Then RQ3 and RQ1 are also answered by utilizing the results of RQ2 and by doing some further evaluation and analysis. The chapter also includes an analysis of the used criteria's validity, quality, and generalizability.

The final step of communication (step 6) was mainly done in the form of this thesis. Also, the results were communicated directly with the stakeholders of Vertex Sync.

5. IMPLEMENTATION

This chapter goes over all the relevant aspects of the third step (design & develop) of the research process. First, the architecture of Vertex Sync and some of the most important design decisions are introduced to give an understanding of why Open Cloud was integrated the way it was. Then the resulting solution is given a brief overview to enable analysis of the solution so that the research questions can be answered comprehensively. Finally, some future improvements are provided for the solution if it is to be iterated upon.

5.1 Overview of the implementation environment

Vertex Sync's architecture follows the microservices architecture. In addition to the different microservices Vertex Sync also has a few other components in the complete system. Vertex Sync is still very new and heavily under development so the architecture is not the definite one. A high-level diagram of the architecture can be seen in figure 5.1. The diagram is based on the architecture of a very early development version of Vertex Sync.

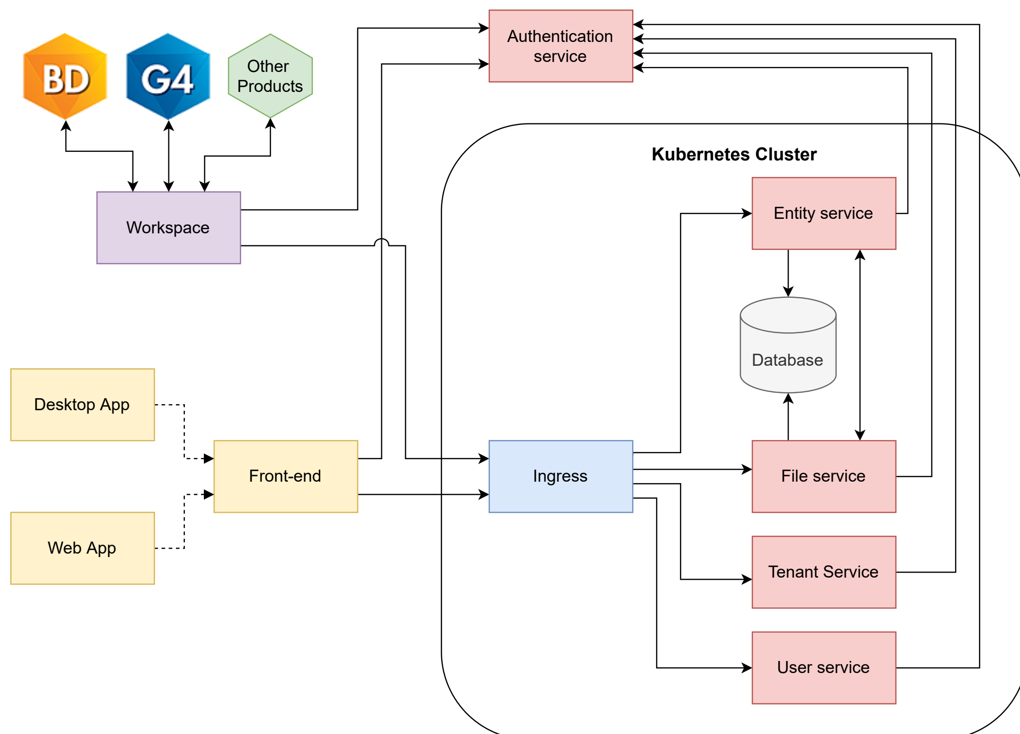

Figure 5.1. *High-level architectural overview of Vertex Sync's early development version.*

The system can be roughly divided into the following categories: main services, front-end, databases, and the Workspace. The following paragraphs briefly describe the roles of the different components.

The front-end consists of an Angular-based SPA and a desktop application based on the Angular application. The front-end communicates with the back-end only via the back-end's REST API, so the front-end can be developed independently from the back-end.

The Workspace is a separately developed component intended for integrating other products into Vertex Sync. It will natively support some of Vertex Systems' own CAD products. Support for external CAD products is planned.

The microservices consist of five different services: authentication, tenant, user, entity, and file services. The authentication service is responsible for all user and service authentication-related issues. It is separately deployed from all the other services. The tenant service is responsible for handling the multitenancy aspect of Vertex Sync and the user service is responsible for handling all user-related issues for each tenant.

Everything that a user uploads to Vertex Sync is represented by an entity. Entities can for example represent files or data, but also collections of other entities. The entity service is responsible for entities and their connections. Entities that represent a file have the actual file inside the file service. The file service is responsible for only storing and managing individual files and their metadata. The file service is exclusively managed by the entity service. Files can be created, modified, or deleted only via the entity service.

All components of the system are deployed with the help of Docker. Docker allows applications to be easily containerized enabling easy deployment in any environment [47]. The individual docker containers in the system are managed using Kubernetes. Kubernetes provides the system with load balancing and service discovery, secret and configuration management, and many other benefits [48].

Communication in Vertex Sync happens through basic HTTP requests. All requests to the main services require the user to have been authenticated via the authentication service. Services communicating with each other also require the requesting service to have been authenticated. All requests from outside the Kubernetes cluster go through Kubernetes Ingress (an API object for managing external access within Kubernetes).

5.2 Major design decisions

As the integration of Open Cloud into Vertex Sync was done in an exploratory manner to see whether it was a good fit for the system or not, it was done in a way where **the existing services required a minimal amount of changes**. This enables Open Cloud to be easily discardable if it was deemed unsuitable for Vertex Sync. This way of integrating

Open Cloud also follows the core values of the microservices architecture well; Open Cloud and the rest of the services will be more loosely coupled and more independent.

All the communication within Vertex Sync happens through traditional HTTP requests. Therefore any communication from existing services to any new components required adding the requests to the source code of the existing services. This new code will also have to be manually removed if Open Cloud is removed from the system. By utilizing some more complex messaging protocols, such as message buses, the existing services would not need as many changes. The existing services could just emit a general message to the bus that any new services could simply subscribe to. This would make the services completely independent of each other and would require no extra work in the existing services if Open Cloud is discarded. However, such mechanisms are only in the planning stages for Vertex Sync and they were not ready to be utilized. Therefore **HTTP requests were used** in the implementation, although some other messaging protocols could have been a better fit. The choice of communication protocol does not have any effect on the suitability of Open Cloud in a microservice-based platform.

One major benefit of Open Cloud for Vertex Sync was its support of BCF. Vertex Sync might require BCF support in the future, so Open Cloud was implemented in a way where **BCF support was kept available**. This required integrating the whole Open Cloud server to Vertex Sync. Otherwise only working with the front-end components and ODA's file converter could have been preferable as the server has many drawbacks.

The use case for Open Cloud within Vertex Sync is centered around user interactions. The user might want to view some CAD files, or in the future, they might want to work with BCF. Both of these use cases are completely covered by Open Cloud, so **Open Cloud only has to be integrated into the system and it does not need to be expanded**. Therefore adding only one new service that is responsible for communicating between Open Cloud and the rest of the services was needed.

The technology-related decisions were based on the implementation of the existing services. The new service did not have any specific technical requirements as long as the technologies were ones that are commonly regarded as good options for making a scalable and responsive HTTP server. The main technologies were therefore chosen based on the existing services so the developers can easily work on different services as necessary. **The new service was implemented using Java** with the Spring WebFlux framework. Spring WebFlux is a free reactive web framework that can take advantage of multi-core processors to handle large amounts of concurrent connections [49].

The front-end part of the implementation had to be merged with the existing front-end, so the technology used to develop the rest of the front-end was a natural choice. **The chosen front-end framework was therefore Angular**. Angular is a free HTML and TypeScript (JavaScript) based web application framework developed by Google. It allows

the creation of performant single-page applications that can be easily deployed for both mobile and desktop devices and as a web or a native application [20].

Open Cloud requires the user to be authenticated to it to serve most requests. Vertex Systems wanted to integrate Open Cloud into Vertex Sync seamlessly for the user so that the user cannot see any separation or distinction between Open Cloud and the rest of the system. This required **the authentication to be handled in a way where the user does not need to know about its existence**. This also meant that extra **effort had to be put into hiding the implementation details**, especially during exceptions.

As the goal of this thesis for Vertex Systems is to see whether Open Cloud is a viable option for Vertex Sync, the development process was focused on getting the system to a workable level. This meant that user experience, UI design, (automated) testing, and other non-essential parts of the normal development process were neglected. If Vertex Systems decides to keep Open Cloud, many parts of the implementation need to be redone to match the normal development standards of Vertex Systems.

5.3 Implementation details

5.3.1 General overview of the implementation

The new implementation resulted in three distinct components to the system: The Open Cloud server, the JobRunner(s), and a new visualizer service. The Open Cloud server and JobRunners were deployed without any extra configuration or modification besides the use of Docker for containerization. The new visualizer service exists as an intermediary between the Open Cloud components and the rest of the services. A visual high-level architectural overview of the implementation can be seen in figure 5.2.

The Open Cloud server requires a MongoDB database in its implementation. Vertex Sync already utilized MongoDB in some other parts of the system, so no additional instances of it needed to be deployed.

In the implementation, Open Cloud was made to utilize the file system for storing data. Open Cloud supports Microsoft Azure BLOBs and Amazon S3 for file storage, so a more scalable implementation should probably use them instead. They will be considered for Vertex Sync if Open Cloud is decided to be kept.

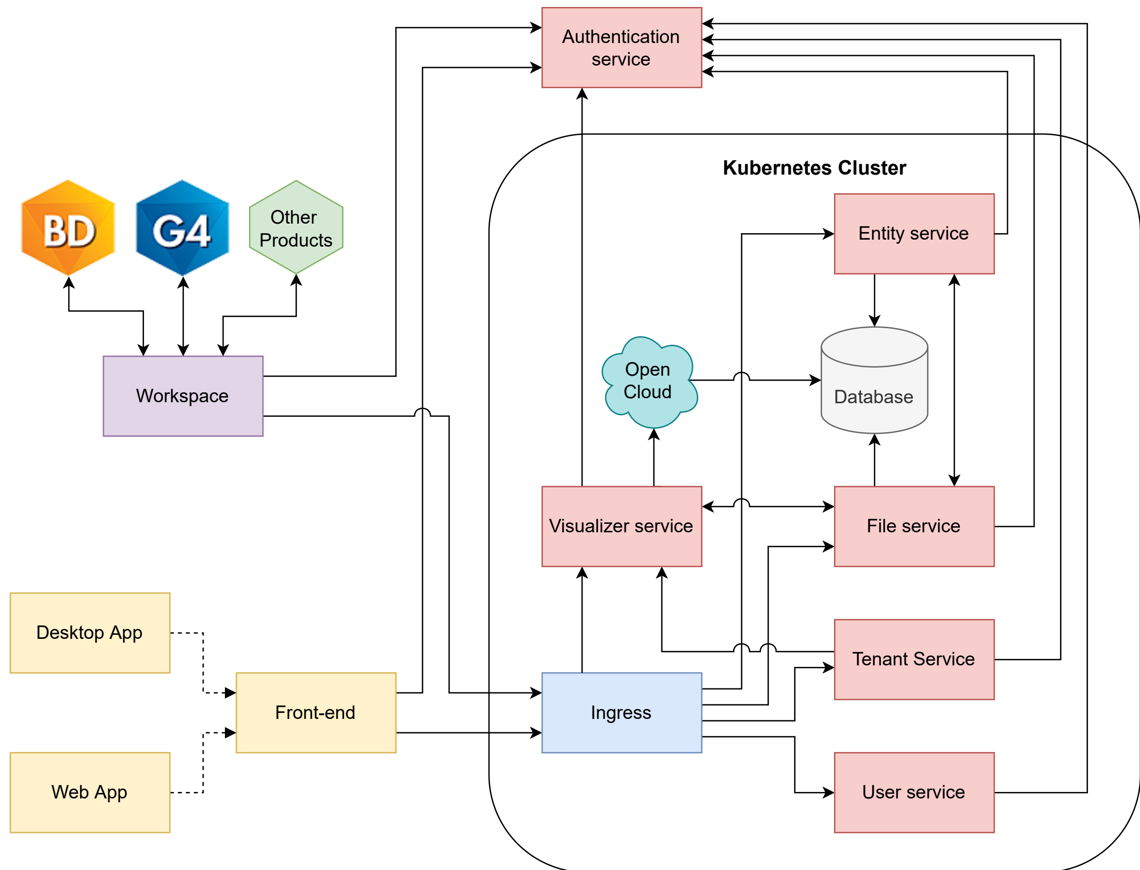

Figure 5.2. *High-level architectural overview of the new implementation.*

5.3.2 Overview of the new visualizer service

The only way to communicate with Open Cloud within Vertex Sync is through the visualizer service. The visualizer service's main responsibilities are to handle Open Cloud's authentication and to communicate between the other services and Open Cloud. Its functionality can be likened to a mix between the mediator from the Gang of Four's software design patterns [50] and the decorating collaborator from Newman's book [5].

All the endpoints of the visualizer service require the client to be authenticated to Vertex Sync. Open Cloud however uses its own authentication system and all of Open Cloud's endpoints require the client to be authenticated to it instead. The way the visualizer service works around this issue is detailed in subsection 6.2.1.

All details about Open Cloud and its implementation should be hidden from the users. This was solved by making all communication to Open Cloud go through the visualizer service. Whenever a client requests something from Open Cloud through the visualizer service, the service makes its own request to Open Cloud. The response from Open Cloud is parsed and all relevant data are transmitted back to the client while unnecessary information, such as error messages, is stripped away from Open Cloud's response. Unwanted information mostly existed in Open Cloud's response headers and error mes-

sages so hiding them was a simple task.

Any errors received from Open Cloud have to be logged and then converted into a more general error message for the client. For example, if a user failed to authenticate to the Open Cloud server the normal response would be a 401 unauthorized error. However, the visualizer service is responsible for authenticating to Open Cloud while the user has to only authenticate to the service. Therefore the error comes from an internal error within the visualizer service, so the correct response for the user's request would be a 500 internal server error. Any errors received from Open Cloud are logged so that the original cause can be examined properly by the developers as necessary.

Without taking error handling and Open Cloud's authentication into account, almost all of the visualizer service's endpoints are simply forwarding requests to Open Cloud. The only exceptions to it are the file uploading endpoint and the GET user endpoint. Details about the file uploading logic are detailed in subsection 6.2.2 and the GET user endpoint in subsection 6.2.4.

5.3.3 Other changes within the system

In addition to the new components, some existing parts of the system had to be modified. The tenant and file service received some fairly minor changes and the front-end mostly received new features related to viewing files with Open Cloud. The number of changes in the back-end could have been much smaller if other communication methods, such as message busses, were used. The services could have sent general messages that the visualizer service subscribed to. With HTTP it required adding several direct calls to the visualizer service from the other services.

The tenant service required only two very small changes. During the creation of a tenant, the service creates a password for the tenant to Open Cloud and notifies the visualizer service of it. This was necessary to handle Open Cloud's authentication. Subsection 6.2.1 will describe this in more detail.

The file service required multiple small changes and a new endpoint. The changes were mostly related to notifying the visualizer service about specific events. One bigger change was that the metadata for files stored in the file service received two new fields for keeping track of the file in Open Cloud. The purpose of the new fields and the new endpoint is detailed in subsection 6.2.2.

The front-end mostly received unnoteworthy changes. The Client.js module had to be installed, and some basic new views, functions, etc. were added for interacting with Client.js. Also, some basic functionalities of the viewer, like camera controls, object properties, and hiding, were implemented for demonstration and testing purposes, and to see what it's like to work with Client.js and VisualizeJs. The issues faced when working with

the libraries are described in subsection 6.2.4.

5.4 Future improvements

5.4.1 Improving usability via notifications

The biggest user experience issue related to the implementation of the system has to do with the responsiveness of the system. When a user starts the file conversion process, the user sometimes has to wait a very long time until the file is viewable. The progress of the conversion cannot be seen at all. When the conversion process finishes, the Open Cloud server just updates the status of the relevant file without sending any notifications. The only way to see the updated status is to query the server randomly to check whether the status has been updated.

To send notifications when a specific job starts or finishes, the file conversion jobs could be expanded by adding a custom job around the conversion job. The custom job could send notifications at the start and the end of its lifespan. When the new job is started, it would first send a notification to the user and then start the original file conversion job. After the file conversion job finishes, the new job sends another notification to the user and then finishes itself.

A new notification service component is under development for Vertex Sync. In the future, the custom job could simply call one of the new service's endpoints and do nothing else. The service could then notify the front-end via a WebSocket connection about the status of a job. It could also send the user that started the job an email about the job finishing.

5.4.2 Smarter communication methods

Vertex Sync only supported the HTTP protocol as the communication method between the different services. A more complex communication protocol, such as a message bus, would have been a better fit for the implementation. In the current implementation, the services have to explicitly call each other. If for example message buses were added, the services could just send general messages that the interested services can subscribe to. This would add looser coupling between the services and lower the technical debt of the system.

Other communication methods in Vertex Sync are under development. Depending on which methods are implemented, all points of communication related to the visualizer service should be inspected and possibly refactored. The current implementation's communication methods have no direct impact on the system, but they should be refactored to improve the maintainability of the system.

5.4.3 Refactoring the implementation

Implementing Open Cloud to Vertex Sync was done as a proof of concept. The development was done quite hastily without putting proper effort into non-essential aspects of the system such as user experience and automated testing. Therefore the entire implementation should be reviewed, some parts refactored and automated testing added to all parts of it.

The back-end part of the implementation is adequate. If automated testing is added and the other problems introduced in section 6.2 have been solved or at least reviewed, the implementation can be used almost as-is. The front-end however needs major reworking. The UI needs to be properly designed, automated testing needs to be added for each component, and the architecture of how the features of the viewer in the front-end are implemented should be redesigned.

6. OBSERVATIONS ON THE IMPLEMENTATION

To be able to evaluate whether the implemented method used to integrate Open Cloud into Vertex Sync can be generalized into a common solution for RQ1, it is first necessary to evaluate the implementation in general. In this chapter individual aspects of the implementation related to the evaluation are introduced and briefly analyzed. They serve as a basis for the fifth step (the evaluation) of the research process done in chapter 7.

6.1 Overview

For the method used to integrate Open Cloud into Vertex Sync to be a generalizable solution for RQ1, the method must be an effective way to integrate Open Cloud into Vertex Sync. The effectiveness of the solution can be evaluated more easily by first evaluating the suitability of Open Cloud for existing microservice-based cloud platforms, as its results can be used in the evaluation of the solution's effectiveness.

The suitability of Open Cloud for existing microservice-based cloud platforms is not a trivial thing to determine. There does not exist any definition as to what is required out of a monolithic service for it to be suitable for a microservice-based cloud platform. In relevant literature, the research is focused on making a monolithic service more suitable for a microservice platform rather than just directly integrating it. For example, many sources focus on describing how a monolithic web service can be migrated into microservices [1, 2] and evaluating whether the migration is worth it [3, 4]. They however focus on how to decompose monoliths and whether the decomposition should be done. With Open Cloud the monolith has to be merged as-is.

In this thesis, the suitability of Open Cloud for existing microservice-based cloud platforms is derived from the implementation by generalizing the results. There are two main factors that decide whether Open Cloud is suitable or not:

1. **Issues found during development.** Section 6.2 includes information about the issues found during development. The issues are described and ways to solve or mitigate them are described as possible. Depending on the severity of the issues and whether they could be solved, they may have a major impact on the suitability of Open Cloud. The impact of each issue on the suitability of Open Cloud is evaluated

in section 6.3.

2. **Fit for microservices.** The implementation must follow some of the core characteristics of microservices to be well suited for a microservice-based cloud platform. Also, some aspects such as the scalability of microservices that follow from their characteristics must be applicable to the implementation. The implementation's fit for microservices is evaluated in section 6.4.

The individual observations related to the criteria are further analyzed in chapter 7 to draw conclusions about the research questions. In addition, the two criteria are further analyzed and generalized to evaluate their generalizability and validity.

6.2 Significant issues found during development

6.2.1 User authentication

User authentication was one of the largest issues faced during development. It is also one of the biggest reasons why another service was needed in between Open Cloud and the other services. Without handling user authentication through a separate service the only other options would be to have public files in Open Cloud, or for users to have two distinct accounts within Vertex Sync. A final option would be to use Open Cloud as the central authentication system, but it was not an option with Vertex Sync.

User authentication was handled in a way where each tenant within Vertex Sync has their own user in the Open Cloud server. All models belonging to a tenant can be seen by all the users of that tenant. The authentication mechanism could be extended to be user-specific, but tenant-specific authentication was deemed good enough for the prototype. A user specific-solution should be considered for a proper implementation.

The user authentication issue was solved with the help of the authentication and tenant service. Whenever a new tenant is added to Vertex Sync, the tenant service adds accounts for each of the services into the authentication service. During the creation of the visualizer service's account, an additional step was added that generates and adds a unique password for that tenant to the authentication service. After the tenant has been fully added, the tenant service notifies the visualizer service of the new tenant. The visualizer service then fetches the newly generated password from the authentication service and creates a new user to Open Cloud using the ID of the tenant and the fetched password. This process is visualized in figure 6.1.

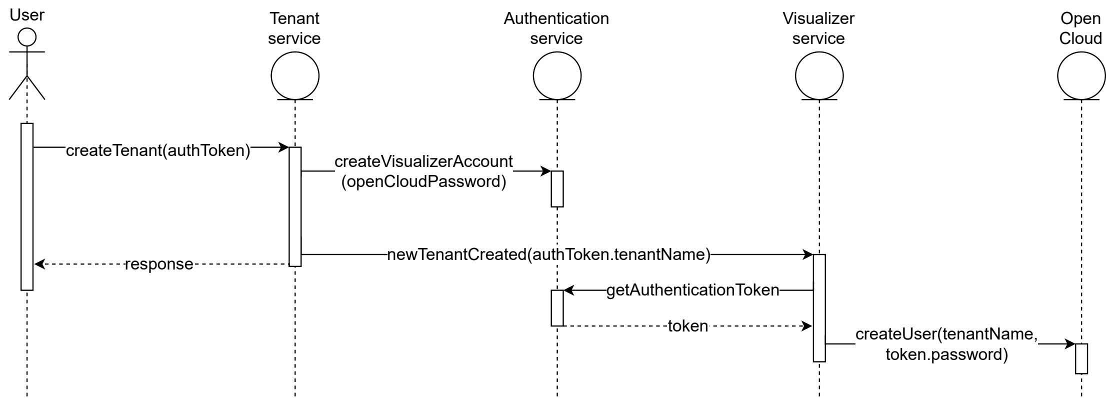

Figure 6.1. *Sequence diagram of tenant creation.*

Afterwards, whenever a request is made to the Open Cloud server through the visualizer service, the visualizer service first fetches the password matching the client's tenant from the authentication server. After fetching the password, the visualizer service authenticates to Open Cloud using the password and then calls the endpoint determined by the request using the received authentication token. Finally, the relevant data is parsed from Open Cloud's response and the original request is served. Figure 6.2 depicts this process.

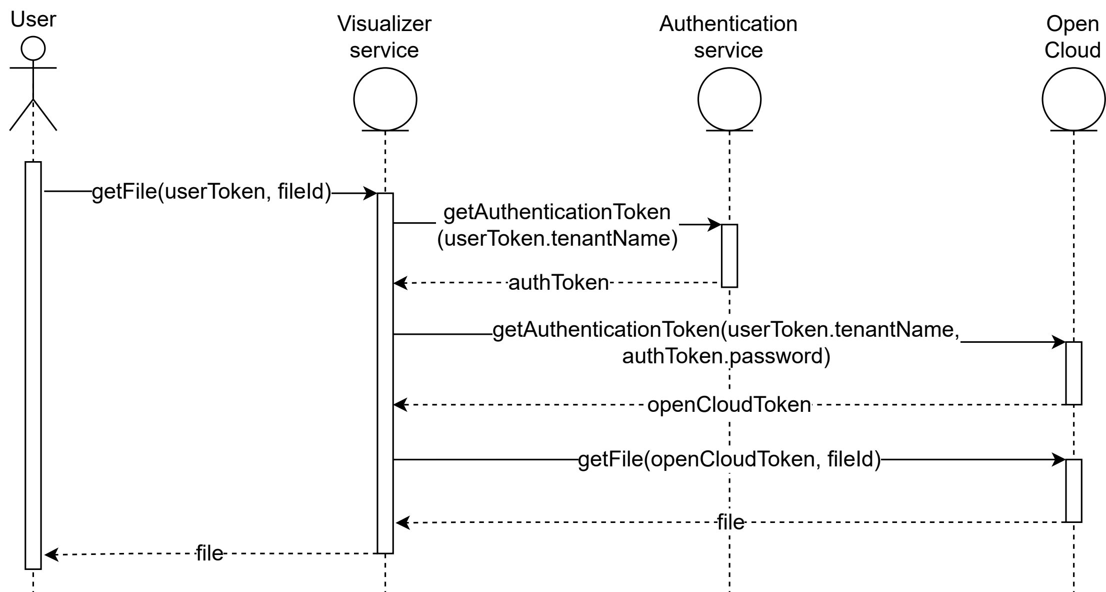

Figure 6.2. *Sequence diagram of a typical request to Open Cloud.*

User authentication to Open Cloud could have also been handled in a simpler way. If the Open Cloud server is only reachable through the visualizer service, all requests from the visualizer service to Open Cloud could be made using Open Cloud's admin token. The client's authorization in Vertex Sync could simply be checked before making the request to Open Cloud. The end result would be the same, but it would pose a much larger security risk. A small programming mistake could easily result in tenant-specific files being accessible by other tenants.

The implemented solution for user authentication works fine, but different users within each tenant can view the files of other users belonging to the same tenant. It does require either brute-forcing or knowing the other user's files' IDs though.

The implemented solution would be quite simple to expand to a user-specific solution. Currently, the accounts in Open Cloud have the tenant name as the username and a randomly generated password managed by the authentication service. A user-specific solution could use the same system, but instead of using tenant names, the username could be the user's name in Vertex Sync. The password could for example be the tenant password appended to some user-specific, permanent data.

Directly storing the user's Open Cloud passwords in the authentication system is not an option. Otherwise, the user could get access to his Open Cloud username and password without the help of the visualizer service. Combining a user-specific password with the tenant-specific password in the visualizer service could be a viable solution.

6.2.2 Uploading files to Open Cloud

Uploading a file to Open Cloud in Vertex Sync should not affect the user at all and the file service should be affected minimally. The file service's file uploading endpoint being the only way of uploading a file to Open Cloud is not an option. In case something went wrong with the upload to Open Cloud (like a service outage), the file upload to Open Cloud should be possible to be restarted by a user. The only way to fulfill these requirements is to run the entire file uploading process inside the visualizer service.

The implemented solution was to implement a file uploading endpoint to the visualizer service that only takes an ID of a file service's file as a parameter. The endpoint has to run the actual process in the background and respond to the request that started the process almost immediately. This way both the user through the front-end and the file service after finishing a file upload only have to do a short request notifying the visualizer service of a new file.

When a client requests a file to be uploaded to Open Cloud, the visualizer service immediately responds with a 200 OK to the user. It then requests the file's metadata from the file service. If the visualizer service deems the file suitable (valid format and not yet uploaded to Open Cloud), it immediately starts downloading the file from the file service. After starting the file download, the visualizer service starts immediately streaming the file to Open Cloud's file uploading endpoint. After the file has been fully uploaded to Open Cloud, the visualizer service makes requests to Open Cloud to extract geometry and properties from the uploaded file. Finally, the visualizer service calls the file service's new endpoint to set the file's status and ID in Open Cloud. A slightly simplified sequence diagram of the file uploading process is visualized in figure 6.3.

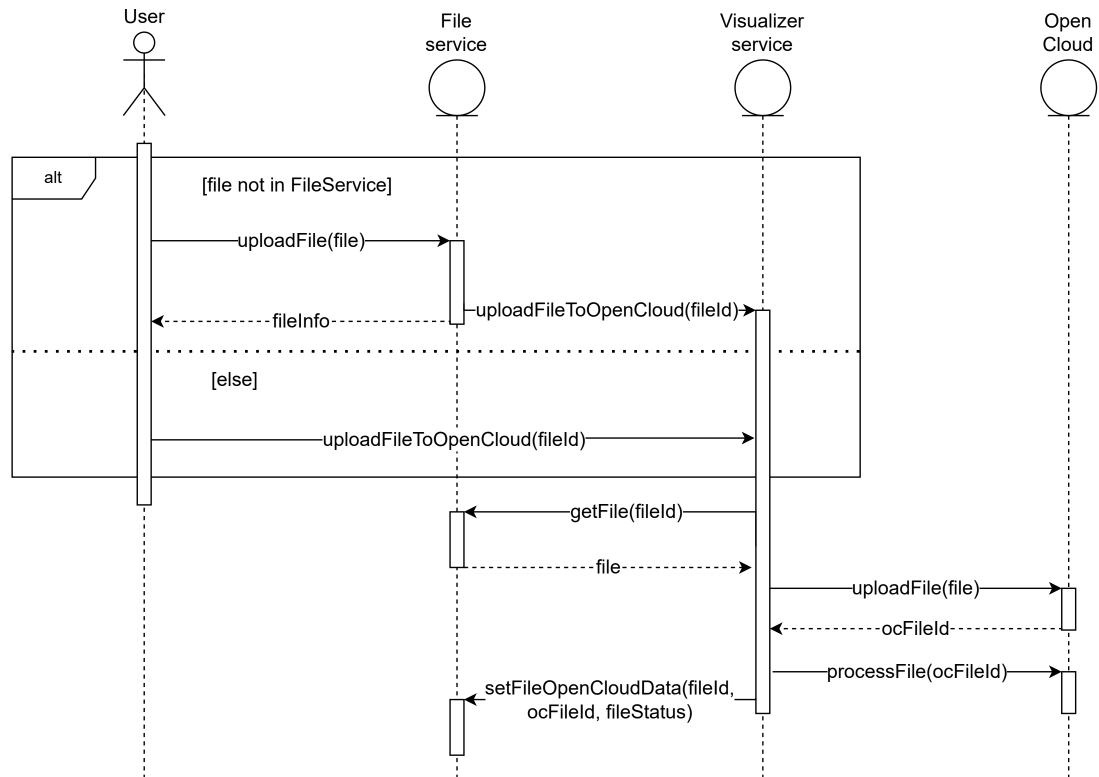

Figure 6.3. *Sequence diagram of uploading files to Open Cloud via the visualizer service.*

In actuality, the file uploading process is significantly more complex. During different parts of the process, the file's metadata in the file service receives differing statuses depending on the upload process's status. Also, the process has several different fallbacks for handling errors in differing parts of the uploading process.

6.2.3 File storage

The largest unsolved issue related to the integration of Open Cloud is related to storing files. All files in Vertex Sync have to be stored in the file service for the normal functionality of the system. Whenever a CAD file has to be viewable, it also needs to be uploaded into Open Cloud. This causes the file to be stored in two different locations. Open Cloud also converts the file into another format, causing a total of 3 different versions of the file to exist in the system. This causes a lot of extra storage space to be consumed causing extra expenses for Vertex Sync.

The converted file in Open Cloud and the original file in the file service are clearly necessary. The copy of the original file in Open Cloud seems unnecessary after the conversion has been done, but Open Cloud does not provide any direct way of deleting only the original file. Support messages have been sent to Open Design Alliance regarding this issue, but for now, there does not exist a solution.

If all uploaded CAD files are automatically added to the Open Cloud server and then

converted, the amount of extra storage space required can cause significant storage-related expenses in Vertex Sync. Removing the duplicate file from Open Cloud is not yet possible, so other means for saving storage space should be considered.

One possibility for saving storage space is to make the files in the Open Cloud server expire. If for example a file is not viewed for a month, it is automatically removed from the Open Cloud server. If a user wanted to view the file again, they could just upload the file to the server with the visualizer service's file uploading endpoint that only takes in the file's ID from the file service. It would harm the general user experience of the system, but it could save a significant amount of storage space especially if Vertex Sync is to be used for years to come.

Open Cloud does not provide any such functionality out of the box, but it would be somewhat trivial to implement. The file service would need to keep track of when a file was last opened. Then a special script or an endpoint could be used that goes through all the files in the Open Cloud server and checks when each file was last opened. If the duration since it was last opened was long enough, the script would delete the file from the Open Cloud server via the visualizer service. The script/endpoint would need to be automatically executed every once in a while (for example once a month).

Another smaller storage usage improvement would be to automatically delete all orphaned files from the Open Cloud server. In Vertex Sync, each file in the Open Cloud server is related to a file in the file service. Normally whenever a file is deleted from the file service, it is automatically also deleted from the Open Cloud server. Under exceptional circumstances, a file from the Open Cloud server might not get deleted even if the one from the file service does. Deleting such files would require keeping track of the file service's file ID in the visualizer service or the Open Cloud server. Open Design Alliance support has been contacted about adding custom data fields to individual files' metadata. Similar custom data fields already exist for users and projects in Open Cloud.

6.2.4 Client.js related issues

Client.js caused multiple minor issues during development. Many of them were solved and they mostly just made the initial development process harder. Many of the issues were ultimately caused by lacking or unclear documentation. Also, some of them were fixed by Open Design Alliance over time.

The authentication system also caused issues with Client.js. When initializing Client.js it requires either an API token or the user's Open Cloud username and password. Immediately after receiving either one, it makes a GET request to the user endpoint of the Open Cloud server. In Vertex Sync the API key could not be made public and the Open Cloud user needed to be hidden so the normal workflow was not possible. The issue was

solved via the visualizer service. First, the Client.js initializer is fed a fake API token. It then tries to fetch a user from Open Cloud, but as all requests to Open Cloud go through the visualizer service in Vertex Sync, the visualizer service receives the request instead. The visualizer service responds to the request with fake user data instead of forwarding it to Open Cloud, unlike almost all the other requests. Afterwards, Client.js can be used normally.

Most requests made by Client.js needed an authentication header for the visualizer service. Client.js does not allow setting authentication headers explicitly. The headers were added by intercepting the fetch requests made by Client.js in the front-end and then adding the headers to the request. It is possible to set the authentication header's value that Client.js sends without using an interceptor. It requires manipulating the user data that Client.js fetches during initialization. The authentication header set this way cannot be updated though, so the method does not work if the authentication token can expire.

At the beginning of development, the TypeScript type declarations included in the Client.js module were faulty. They could not be used in Angular even though they existed in the module. It was caused by an issue within the Client.js module that Open Design Alliance fixed during the development of this thesis.

The type declarations for Client.js are also somewhat lacking. Using Client.js requires working with VisualizeJs most of the time. VisualizeJs however does not have any type declarations and Client.js does not add them. This caused many of Client.js's type declarations to be marked as "any", even though they had a clear type within VisualizeJs. The typings were improved in Vertex Sync by extending the type declarations of Client.js. Extending the typings was also necessary as many of the types of Client.js were slightly faulty. For example, nearly all primitive types in Client.js were faultily typed using their object-wrapped variants (boolean vs. Boolean).

Also, the documentation of the front-end components is somewhat lacking. Many functions have no description at all, just the parameter and return types (also without description). The lacking documentation for VisualizeJs was somewhat excusable though, as it is heavily based on ODA's Visualize SDK, which has extensive documentation. There do exist some differences between VisualizeJs and Visualize SDK. For example, some function names can differ which can make finding the relevant information much harder.

A great example of what the lacking documentation caused during development is related to the enableSceneGraph function. All it has for documentation in Open Cloud is just the name of the function. A proper description could be found from Visualize SDK's documentation, but only under general device options. According to the documentation it "can increase memory usage by up to two times but rendering performance by up to 10 times". The setting was used in the Open Cloud demo application and it caused significant confusion as to why the demo's performance was so much better than Vertex

Sync's. The source of the differing performance was found by going through the demo implementations source code line by line inspecting every unfamiliar function call.

In the end, Client.js (and VisualizeJs) worked well overall with only minor issues making development harder. Everything could be fixed or worked around, and Open Design Alliance fixed the issues that were reported to them. The issues related to Client.js and the documentation do undermine the credibility of Open Cloud though. The TypeScript related issues were quite obvious if one were to use them. The TypeScript related parts must have been not tested at all, or at least insufficiently, so it creates doubts whether Open Cloud has been properly tested overall.

6.3 Impact of the issues on the suitability of Open Cloud

The impact of the issues found during development on the suitability of Open Cloud varies a lot depending on the issue and how well it could be solved. The user authentication issue can have a very significant impact on the suitability of Open Cloud if it is not solved well. The implemented solution of handling Open Cloud's user authentication with the help of an intermediary service works well. If user-specific Open Cloud accounts were used instead of the current tenant-specific ones, the impact of the issue would be almost nonexistent. It does slow down requests to Open Cloud a bit, but it can be optimized significantly by caching the results of previous requests.

The file uploading process has a moderate impact on the suitability of Open Cloud. The implemented file uploading process works well. It is reliable and it hides the issue from the user. It however has a significant impact on the system's performance and resource consumption, causing a moderate impact on Open Cloud's suitability.

The file storage issue also causes a moderate impact on the suitability of Open Cloud. The need to store duplicates of a file causes extra expenses to the operation of the system. It can however be alleviated a little via the methods described at the end of subsection 6.2.3. Also if Open Design Alliance enables the possibility to delete the original files after a CAD file has been converted, the issue could be avoided almost completely.

The Client.js issues do not have any impact on the suitability of Open Cloud. They mostly make development harder and they harm the credibility of Open Cloud as it raises doubts about its quality.

6.4 The implementation's fit for microservices

For Open Cloud to be suitable for a microservice-based cloud platform, it needs to fit the nature of microservices. In particular, it needs to follow the core characteristics of microservices introduced in subsection 2.2.1. The following paragraphs describe the im-

plementation's fit for each of the introduced characteristics and their impact on Open Cloud's suitability:

Small size Open Cloud itself is not small. Its functionality is however clearly separated behind different REST resources so it is still easily understandable. Also, if Open Cloud is only accessible via another service, the service could be easily split into multiple smaller services each responsible for one area of Open Cloud. So even though Open Cloud nor the implementation is of small size, it is not an issue for the suitability of Open Cloud.

Flexibility Open Cloud is not flexible at all. It requires its own authentication system and file storage and not much can be configured about it. By having another service in between Open Cloud and the rest of the system, the most problematic parts of Open Cloud can be worked around, increasing its' flexibility. However, it is still not flexible and it has a minor impact on the suitability of Open Cloud.

Independent deployability The Open Cloud server and the related components can all be deployed independently. It does not harm the suitability of Open Cloud.

Modeled around a business domain Typically microservices are responsible for only one business domain. Open Cloud however has several business domains it is responsible for (for example file storage and BCF), but they can be easily represented via multiple smaller services. It is not an issue for the suitability of Open Cloud.

Autonomously developed & Alignment of architecture and organization Open Cloud can be developed autonomously. Only the developers responsible for working around Open Cloud's features need to know about its existence, so it is not an issue for the suitability of Open Cloud.

Decentralized The implementation is decentralizable as all parts of it are containerized with Docker and deployed with Kubernetes. Decentralizability is therefore not an issue for Open Cloud.

For Open Cloud to be suitable for microservices-based cloud platforms, it also has to bring in similar advantages as normal microservices do. Of the advantages of microservices introduced in section 2.2.2, the implementation brings in all of them. The implementation is scalable by adding more JobRunners. The implementation is technology agnostic as it only communicates via HTTP. It is fault-tolerant & resilient as errors or the loss of the visualizer service or Open Cloud has no impact on the rest of the system. The implementation is mostly reusable by simply removing the Vertex Sync-specific logic out of it. It is also quite easy to understand as Open Cloud has a simple API and the visualizer service's purpose is clear.

7. DISCUSSION

This chapter represents the fifth step (the evaluation) of the research process. The individual observations from chapter 6 are combined and analyzed to answer the research questions. In addition, the criteria used in the evaluation of Open Cloud's suitability are further discussed and analyzed to provide knowledge about their validity and generalizability.

7.1 RQ2: Is Open Cloud suitable for existing microservice-based cloud platforms

In this thesis, Open Cloud's suitability for existing microservice-based cloud platforms is determined by the implementation's fit for microservices and the issues found during development. Table 7.1 recaps the issues found during development from section 6.2 and table 7.2 recaps the implementation's fit for microservices from section 6.4. The impact of the issues is evaluated with a scale of none, minor, moderate, and major. In table 7.1 the impact is separated into potential and realized impact. The potential impact refers to the impact of the issue if no effort is put into solving it and the realized impact evaluates the impact with the implemented solution. The table 7.2 also includes a column that describes how well the implementation fits each microservice characteristic with a scale of poor-adequate-good.

Issue	Potential impact	Status	Realized impact
User authentication	Major	Solved	None
File uploading	Major	Solved	Moderate
File storage	Moderate	Unsolved	Moderate
Client.js issues	Minor	Solved	None

Table 7.1. *Impact of the issues found during development*

Characteristic	Fit	Impact
Small size	Adequate	None
Flexibility	Poor	Minor
Independent deployability	Good	None
Modeled around a business domain	Good	None
Autonomously developed	Good	None
Alignment of architecture and organization	Good	None
Decentralized	Good	None

Table 7.2. *The implementation's fit for microservices*

In addition to the microservice characteristics, the implementation was compared to the advantages of microservices. The implementation was deemed to have all the same advantages as the ones listed for microservices in section 2.2.2.

The implementation therefore has two moderate and one minor problem related to its suitability. The two moderate issues mainly cause performance issues in the implementation. The file uploading issue causes extra network traffic and read/write operations and the file storage issue causes significant extra file storage usage. Both of the issues cause extra expenses in running Open Cloud making it less suitable, but alone they do not make Open Cloud unsuitable. Also, it may be possible to improve the file uploading process, and ways to improve the file storage issue are being looked into.

The implementation fits the nature of microservices very well. The only thing the implementation is missing is the flexibility of normal microservices. It however is only a minor issue, which does not make Open Cloud unsuitable.

In conclusion, Open Cloud is fairly well suited to be used in existing microservice-based cloud platforms. It however requires the use of an intermediary service to work around some of the more severe issues that would make Open Cloud unsuitable. It is also not ideal in terms of resource consumption.

7.2 RQ3: Is Open Cloud suitable for Vertex Sync

As the evaluation of the suitability of Open Cloud for Vertex Sync is of less interest to the general public and it relies heavily upon the subjective evaluations of the people responsible for Vertex Sync, it was done privately within Vertex Systems. The evaluation was fairly unstructured and it consisted of freely testing the implemented system, a performance analysis of the implemented solution's back-end, and a comparison of the front-end's performance to other systems. Also, the results and findings of section 7.1 were presented

and kept in mind during the evaluation process.

The result of the evaluation is that Open Cloud is suitable for Vertex Sync, however, it is not very ideal. The biggest drawbacks of Open Cloud for Vertex Sync were related to file storage, Client.js, and the extra expenses the two issues caused.

Vertex Sync is a multitenant system where the tenants are kept completely isolated. Open Cloud has no tenant concept, so all the tenants' data within Open Cloud is mixed together. This causes slight risks of tenant-specific data leaking to other tenants. Also, the data within Open Cloud cannot be handled tenant-specifically. For example, the data within Open Cloud cannot be rolled back for only one tenant.

The Client.js-related issues were mostly an issue for Vertex Sync because of credibility issues. The lacking documentation and the severely faulty typings in Client.js raised serious doubts about Open Cloud's quality. The doubts also raised concerns about the security of the Open Cloud server. In the end, the doubts about security were accepted as Open Cloud could be completely isolated from the outside world within Vertex Sync. The visualizer service can be kept as the only access point to the Open Cloud server.

The increased resource usage caused by the file storage issues and the complex file uploading process caused some concerns about the expenses of using Open Cloud within Vertex Sync. The increased file storage usage might be possible to be improved by utilizing the methods described in section 6.2.3. The extra resource usage of the file uploading process is hard to improve without reworking the entire process. The total resource usage of Open Cloud and the expenses they cause need to be evaluated and compared to the rest of Vertex Sync. This is however outside the scope of this thesis and it will be done separately by Vertex Systems.

7.3 RQ1: How to integrate a monolithic web service into a microservice-based cloud platform without modifying the monolith

The question of how to integrate a monolithic web service into a microservice-based platform without modifying the monolith has not been researched. The method used in this thesis to integrate Open Cloud into Vertex Sync can be quite easily generalized into a common solution for similar situations. Open Cloud should be fairly generalizable to an arbitrary monolithic web service as it features many of the common aspects one would assume a monolithic web service could have, such as separate user authentication and file storage. It is also not missing any vital features monolithic web services often have.

The way Open Cloud was integrated into Vertex Sync using an intermediary microservice to handle all communication between Open Cloud and the other microservices proved to

be a simple and viable option. The same method should be applicable to any monolithic web service and microservice-based cloud platform, however, the solution might not be the ideal one depending on the monolith, the platform, and the use case. An example of a viable alternative solution could be to have one microservice handling all the communication to the monolith that solves the general issues related to the integration and then have separate microservices for each of the different business areas the monolith is used for. The business area-specific microservices can then easily be used to solve issues related to the specific business areas and expand them as necessary. All communication to the monolith would go through the business-area specific services that contact the main intermediary microservice as necessary. The business area-specific microservices could then be easily swapped out, deleted, or refactored to not use the monolith at all.

Sam Newman's decorating collaborator pattern [5] would not have been a suitable pattern to be used in the integration of Open Cloud. Issues similar to the authentication issue with Open Cloud cannot be solved without adding a more substantial microservice between the monolith and the microservices. Also, in cases similar to Open Cloud it needs to be enforced that all communication to the monolith goes through the intermediary microservice. However, it does not mean that the decorating collaborator is a bad pattern for integrating monolithic web services to microservice-based cloud platforms. With a more simplistic or modular monolithic web service that does not have significant technical issues related to integrating it a decorating collaborator might be an ideal solution.

The individual issues related to integrating Open Cloud introduced in section 6.2 are quite Open Cloud-specific. The Client.js issues are clearly specific to Open Cloud. The file storage issue is also highly Open Cloud-specific, but there is a chance some other system would need to store its files to enable some features similar to Open Cloud. The file uploading issue is reasonably likely to be encountered in other systems if they need files to be uploaded at least temporarily to enable some features. The authentication issue is quite likely to be encountered in other similar systems as many web services contain their own authentication system. The implemented/planned solutions for the three generalizable issues solved the issues relatively well and they can be easily applied to other similar cases.

The way Open Cloud was integrated into Vertex Sync using an intermediary microservice responsible for all communication between the monolithic web service and the other microservices should prove to be a simple and effective solution to the research question. Most common problems related to the integration should be solvable with the used method. However, the ideal way a monolithic web service should be integrated into a microservice-based cloud platform can vary a lot depending on the monolith, the platform, and the use case. The individual problems related to integrating the monolith are also highly dependent on the systems in question, so careful planning should be put into deciding the way a monolithic web service should be integrated into a microservice-

based cloud platform. An intermediary microservice is probably necessary, however its implementation pattern and the need for additional components need to be evaluated depending on the specific system and use case.

7.4 Evaluating the suitability of a monolithic web service for a microservice-based cloud platform

The issue of how to evaluate the suitability of a monolith for a microservice-based cloud platform without decomposition has not been researched either. All relevant literature either focuses on evaluating whether and how the monolith should be decomposed or how well the decomposition succeeded (for example [3, 4, 51]), or evaluating the quality of the services themselves. When evaluating the suitability of a monolithic web service for a microservice-based platform it is important to note that the quality of the services is somewhat irrelevant to the suitability. The common quality attributes, such as performance, scalability, or security, mostly measure the quality of the monolith as an individual product. The monolith's quality as a product is mostly irrelevant to its suitability for a microservice-based cloud platform.

In this thesis, the suitability of Open Cloud was evaluated by analyzing the issues found during development and the implementation's fit for microservices. The criteria gave valuable information about the implementation's suitability. The issues found during development provided essential knowledge about unforeseen issues related to Open Cloud's suitability. The fit for microservices criterion seems important as well as some of the characteristics and advantages of microservices are necessary for the implementation to be suitable for microservice-based cloud platforms. For example, the implementation must be scalable, decentralizable, and independently deployable for it to be suitable at all.

However, the criteria are not comprehensive enough to be used as the sole criteria for the evaluation. Some of the individual aspects of both of the criteria require subjective analysis especially since some of the aspects might not be relevant to the monolith's suitability. They however provide valuable information about the suitability as most issues that could make the monolith unsuitable would be detected by the criteria.

Another issue related to the used criteria is that they can be only used to evaluate an already integrated system. They do not provide any information as to the suitability of a monolith that is only being considered to be integrated into a microservice-based platform.

The subject of how to evaluate the suitability of a monolithic web service for a microservice-based cloud platform without decomposition needs to be researched further. Simple, comprehensive criteria are needed as to what is required out of a monolithic service for it to be suitable for microservice-based cloud platforms. The criteria used in this thesis can be used as a working base, but alone they are not comprehensive enough.

8. CONCLUSION

The purpose of this thesis was to find out how monolithic web services can be integrated into microservice-based cloud platforms without modifying the monolith (RQ1), and to find out if Open Design Alliance's Open Cloud is suitable to be used in existing microservice-based cloud platforms (RQ2) and Vertex Sync (RQ3). To explore and answer the research questions a proof of concept implementation of integrating Open Cloud into Vertex Sync was made, which was then evaluated. Based on the issues found during development and how well the implementation fits the nature of microservices, conclusions were drawn about the suitability of Open Cloud. The question of how to integrate monolithic web services into microservice-based cloud platforms without modifying the monolith was answered based on the results of the other research questions and the method used to integrate Open Cloud into Vertex Sync.

Open Cloud's architecture is that of a monolith. All the features of the Open Cloud server have to be used and barely anything can be configured. Integrating such a server into an existing microservice-based cloud platform requires working around some of the most problematic aspects of Open Cloud, such as user authentication and file storage. In the implementation, the problematic parts were worked around by adding an intermediary microservice between the Open Cloud server and the rest of the system. The new service is responsible for handling and merging Open Cloud into the rest of the microservice system.

During the development of the implementation, two issues were found that had an impact on the suitability of Open Cloud by causing extra resource consumption to the complete system. They were however not significant enough to make Open Cloud unsuitable to be used in existing microservice-based platforms. The implementation was also determined to fit the nature of microservices very well. Therefore Open Cloud was judged to be suitable to be used in existing microservice-based cloud platforms (RQ2). It however requires the use of another microservice to work around the most significant issues related to the integration. The resource consumption of Open Cloud is also not very ideal as the integration causes some resource consumption-related issues.

Vertex Systems judged the implementation to be preliminarily acceptable for Vertex Sync (RQ3) since Open Cloud was deemed suitable for existing microservice-based cloud plat-

forms, Vertex Systems did not find any of the encountered issues too significant, and the performance of the implementation was also deemed acceptable. Vertex Systems still needs to evaluate the costs of running Open Cloud as part of Vertex Sync before it can be accepted for sure.

The implemented method of having an intermediary microservice responsible for all communication between Open Cloud and the other microservices was deemed a simple and effective solution for the question of how monolithic web services can be integrated into microservice-based cloud platforms without modifying the monolith (RQ1). The implemented method should be an applicable solution for the integration of any monolithic service and microservice platform, that can be used to solve most issues related to the integration. However, it should be carefully considered whether additional microservices are needed in addition to the intermediary one depending on the monolith, the cloud platform, and the use case of the monolith in the platform.

Overall the thesis was a success. Answers to all three of the research questions were found and Vertex Sync was provided with a working base for integrating Open Cloud into it.

REFERENCES

[1] Luís Nunes, Nuno Santos, and António Rito Silva. "From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts". In: *Software Architecture*. Vol. 11681. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 37–52. ISBN: 3030299821.

[2] Aleksandra Stojkov and Zeljko Stojanov. "Review of methods for migrating software systems to microservices architecture". In: *Journal of Engineering Management and Competitiveness (Online)* 11.2 (2021), pp. 152–162. ISSN: 2334-9638.

[3] Florian Auer et al. "From monolithic systems to Microservices: An assessment framework". eng. In: *Information and Software Technology* 137 (2021). ISSN: 0950-5849.

[4] Diogo Faustino et al. "Stepwise Migration of a Monolith to a Microservices Architecture: Performance and Migration Effort Evaluation". In: (2022).

[5] Sam Newman. *Monolith to Microservices*. O'Reilly Media, Inc, 2019. ISBN: 9781492047834.

[6] Open Design Alliance. *Open Cloud*. https://www.opendesign.com/products/open-cloud. Accessed on 17.02.2022.

[7] Chris Richardson. *Microservices patterns : with examples in Java*. 1st edition. Shelter Island, NY: Manning Publications, 2019. ISBN: 1-61729-454-3.

[8] Sam Newman. *Building Microservices, 2nd Edition*. O'Reilly Media, Inc, 2021. ISBN: 9781492034018.

[9] Eberhard Wolff. *Microservices*. Addison-Wesley Professional, 2016. ISBN: 9780134602417.

[10] Gaurav Kumar Aroraa. *Building microservices with .NET Core : transitioning monolithic architecture using microservices with .NET Core*. 1st edition. Birmingham, [England] ; Packt Publishing, 2017. ISBN: 1-78588-496-4.

[11] Nabor C Mendonca et al. "The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture". In: *IEEE software* 38.5 (2021), pp. 17–22. ISSN: 0740-7459.

[12] Dinesh Rajput. *Hands-On Microservices - Monitoring and Testing*. 1st edition. Packt Publishing, 2018. ISBN: 1-78913-360-2.

[13] Netflix. *The Netflix Tech Blog*. netflixtechblog.com. Accessed on 11.02.2022.

[14] Cesare Pautasso et al. "Microservices in Practice, Part 1: Reality Check and Service Design". In: *IEEE software* 34.1 (2017), pp. 91–98. ISSN: 0740-7459.

[15] Fowler Lewis. *Microservices: a definition of this new architectural term*. martinfowler.com/articles/microservices.html. Accessed on 10.02.2022. 2014.

[16] Vivek Kale. "Distributed Systems Basics". In: *Parallel Computing Architectures and APIs*. 1st ed. CRC Press, 2020, pp. 65–79. ISBN: 9781138553910.

[17] Irakli Nadareishvili. *Microservice architecture : aligning principles, practices, and culture*. First edition. Beijing, [China: O'Reilly, 2016. ISBN: 1-4919-5632-1.

[18] Mike Loukides and Steve Swoyer. *Microservices adoption in 2020*. https://www.oreilly.com/radar/microservices-adoption-in-2020/. July 2020. Accessed on 15.02.2022.

[19] Emmit A. Scott. *SPA design and architecture : understanding single-page web applications*. 1st edition. Manning, 2016. ISBN: 1-61729-243-5.

[20] Google. *Angular*. https://angular.io/. Accessed on 11.02.2022.

[21] Meta. *React*. reactjs.org. Accessed on 11.02.2022.

[22] Evan You. *Vue*. https://vuejs.org/. Accessed on 11.02.2022.

[23] *Angular documentation*. https://angular.io/guide/what-is-angular. Accessed on 11.02.2022.

[24] *React documentation*. https://reactjs.org/docs/getting-started.html. Accessed on 11.02.2022.

[25] *Vue documentation*. vuejs.org/guide/introduction.html. Accessed on 11.02.2022.

[26] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[27] Herbjorn Wilhelmsen. *SOA with REST : principles, patterns & constraints for building enterprise solutions with REST*. 1st edition. The Prentice Hall service-oriented computing series from Thomas Erl. Prentice Hall, 2012. ISBN: 0-13-701251-9.

[28] *REST: Advanced Research Topics and Practical Applications*. 1st ed. 2014. New York, NY: Springer New York, 2014. ISBN: 1-4614-9299-8.

[29] Pascal Giessler et al. "Best Practices for the Design of RESTful Web Services". In: *Proceedings - International Conference on Software Engineering* 10 (Nov. 2015), pp. 392–397.

[30] Mark Masse. *REST API Design Rulebook*. O'Reilly Media, Inc, 2011. ISBN: 1449317901.

[31] Leonard Richardson. *RESTful Web APIs*. 1st edition. North Sebastopol, California: O'Reilly, 2013. ISBN: 1-4493-5974-4.

[32] *REST: From Research to Practice*. 1st ed. 2011. New York, NY: Springer New York, 2011. ISBN: 1-4419-8303-1.

[33] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. http://www.rfc-editor.org/rfc/rfc7231.txt. RFC Editor, June 2014.

[34] Open Design Alliance. *Open Cloud documentation*. https://cloud.opendesign.com/docs/index.html#/overview. Accessed on 17.02.2022.

[35] Open Design Alliance. *About ODA*. https://www.opendesign.com/about. Accessed on 17.02.2022.

[36] Lisa Kempfer. "Open DWG alliance". In: *Computer-aided engineering* 17.3 (1998). ISSN: 0733-3536.

[37] Open Design Alliance. *Members*. https://www.opendesign.com/member-showcase. Accessed on 17.02.2022.

[38] Open Design Alliance. *ODA Platform*. https://www.opendesign.com/products. Accessed on 17.02.2022.

[39] Open Design Alliance. *BuildingSMART and Oda Announce Strategic Partnership*. https://www.opendesign.com/blog/2019/september/buildingsmart-and-oda-announce-strategic-partnership. Accessed on 17.02.2022.

[40] Open Design Alliance. *Open Cloud Changelog*. https://cloud.opendesign.com/docs/index.html#/changelog. Accessed on 22.02.2022.

[41] Open Design Alliance. *Open Cloud online demo*. https://cloud.opendesign.com/index.html#/. Accessed on 22.02.2022.

[42] Jaap Kabbedijk et al. "Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective". In: *The Journal of systems and software* 100 (2015), pp. 139–148. ISSN: 0164-1212.

[43] Alan R Hevner et al. "Design Science in Information Systems Research". In: *MIS quarterly* 28.1 (2004), pp. 75–105. ISSN: 0276-7783.

[44] John. Gerring. *Case study research : principles and practices*. Cambridge: Cambridge University Press, 2007. ISBN: 978-0-521-85928-8.

[45] Vijay K. Vaishnavi and William Kuechler. *Design Science Research Methods and Patterns, 2nd Edition*. CRC Press, 2015. ISBN: 1498715257.

[46] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *Journal of management information systems* 24.3 (2007), pp. 45–77. ISSN: 0742-1222.

[47] Inc. Docker. *Docker*. https://www.docker.com/. Accessed on 21.02.2022.

[48] Cloud Native Computing Foundation. *Kubernetes*. https://kubernetes.io/. Accessed on 21.02.2022.

[49] VMware. *Spring WebFlux*. https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html. Accessed on 21.02.2022.

[50] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. 37th printing. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995. ISBN: 0-321-70069-4.

[51] Davide Taibi and Kari Systä. "A Decomposition and Metric-Based Evaluation Framework for Microservices". eng. In: *Cloud Computing and Services Science*. Vol. 1218. Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 133–149. ISBN: 3030494314.