Integration of SaaS Source Systems to Cloud Data Warehouse

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Data Analytics
April 2026
Veeti Koivuniemi

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

TURUN YLIOPISTO
Tietotekniikan laitos

Veeti Koivuniemi: Integration of SaaS Source Systems to Cloud Data Warehouse

Diplomityö, 53 s.
Data-analytiikka
Huhtikuu 2026

Liiketoimintaympäristössä, jossa tietomäärät kasvavat jatkuvasti, tietojen tallennuksen merkitys korostuu. Samanaikaisesti palveluna ostettujen ohjelmistojen (SaaS) suosion kasvaessa dataan omistukseen ja käyttöön liittyvät ongelmat kasvavat suorien tietokantayhteyksien puutteen vuoksi. Tämän tutkimuksen tarkoituksena on selvittää, miten SaaS-ohjelmistojen data voidaan tallentaa tietovarastoon käyttäen sovellusrajapintoja (API) ensisijaisena integraatiomenetelmänä.

Tutkimus alkaa pilvipalveluiden, tietovarastoinnin ja sovellusrajapintojen (API) teoreettisella tarkastelulla, tarjoten tarvittavat perustiedot SaaS-ympäristöjen tiedon integroinnin ymmärtämiseksi. Teoreettinen osuus käsittelee myös pilvipalveluiden keskeisiä käsitteitä, tietovarastoinnin kehitystä ja työkaluja sekä REST-pohjaisia sovellusrajapintoja. Lisäksi tutkimuksessa tarkastellaan IFS-toiminnanohjausjärjestelmää, joka toimii tämän tutkimuksen käytännön osassa lähdejärjestelmänä.

REST-rajapintoihin perustuvan tietovarastoinnin arvioimiseksi toteutettiin konseptitodistuksena integraatioputki, jossa tietovarastointiin käytettiin pelkästään IFS:n tarjoamia REST-rajapintoja. Integraatioputki on toteutettu Azure Data Factory -alustalla ja kohdetietovarastona toimii Azuressa sijaitseva tietokanta. Toteutuksessa tarkasteltiin eri skenaarioita, kuten suurten datamäärien siirtoa, konfiguroitujen attribuuttien käsittelyä ja useista eri tietokantatauluista peräisin olevan datan integrointia. REST-pohjaisen toteutuksen suorituskykyä ja rajoitteita analysoitiin ja verrattiin perinteisiin SQL-pohjaisiin toteutuksiin.

Tulokset osoittavat REST-rajapintojen olevan yksinkertainen tapa datan hakemiseen, mutta niiden olevan huomattavasti hitaampia perinteisiin menetelmiin verrattuna. Tämän vuoksi REST-rajapintojen kautta tapahtuva datan lataaminen ei tällä hetkellä välttämättä sovellu suurille datamäärille ilman optimointia.

Asiasanat: Ohjelmisto palveluna (SaaS), tietovarastointi, REST-rajapinta, pilvipohjainen toiminnanohjausjärjestelmä (ERP)

UNIVERSITY OF TURKU
Department of Computing

Veeti Koivuniemi: Integration of SaaS Source Systems to Cloud Data Warehouse

Master of Science (Tech) Thesis, 53 p.
Data Analytics
April 2026

In a business environment characterized by growing volumes of data, the importance of data storage is becoming increasingly evident. At the same time, the adoption of software-as-a-service (SaaS) solutions has introduced new challenges related to data ownership and access, as direct database connections are often restricted. The purpose of this study is to examine how data from SaaS products can be stored in a data warehouse environment, with a focus on using application programming interfaces (APIs) as the primary data integration method.

The study begins with a theoretical background of cloud computing, data warehousing, and APIs, providing the necessary background for understanding data integration in SaaS environments. Theoretical background includes detailed information on key concepts of cloud computing, evolution and tools of data warehousing, and REST-based APIs. We will also examine IFS ERP, which serves as the source system in the practical part of this study.

To evaluate REST API-based data warehousing, a proof-of-concept integration pipeline for data warehousing using only IFS provided REST APIs is implemented. The data integration of the pipeline is built in Azure Data Factory, with an Azure-based database serving as the target data warehouse. The implementation explores multiple scenarios, including large data transfers, handling of custom attributes, and integration of data from multiple different database tables as sources. The performance and limitations of the REST-based approach are analyzed and compared to traditional SQL-based data loading methods.

The results show that, although REST APIs are an easy way to retrieve data, they are significantly slower than conventional methods. For this reason, REST API data retrievals are not necessarily suitable for large amounts of data at this time without optimization.

Keywords: Software-as-a-service (SaaS), data warehousing, REST API, Cloud Enterprise Resource Planning (ERP)

Contents

1 Introduction **1**

1.1 Purpose of the thesis 2

1.2 Research questions 2

1.3 Thesis structure 3

2 Technical background **5**

2.1 Cloud computing 5

2.2 Data warehousing 14

2.3 Application programming interfaces 20

3 Integrating SaaS source systems to DW: IFS case study **25**

3.1 IFS 25

3.1.1 IFS API 26

3.2 Case study 30

3.2.1 Loading a single large table 31

3.2.2 Custom attributes in the target table 35

3.2.3 Transferring data from multiple sources with one load 39

4 Results **43**

4.1 Implementation 3.2.1 43

4.2 Implementation 3.2.2. 45

4.3 Implementation 3.2.3. 47

5 Conclusion **49**

5.1 Limitations 51

5.2 Future work 52

References **54**

1 Introduction

In today's business environment, the value of data has grown to the point where it can be said that data-driven decision making separates the winners from the losers [1]. In general, the process of companies collecting data and turning it into meaningful decisions is called business intelligence [2]. To improve their business intelligence based decision making companies are constantly seeking to collect and use data more efficiently and with new methods.

Data warehouses are one of the main ways for companies to maintain data for business intelligence tools and other analytical processes to gain information about the company's processes. This allows companies to create data-driven decisions instead of decisions based on intuition [3]. Data warehouses are a powerful tool for business intelligence because they integrate data from several different data sources in to one integrated location [4]. This allows analysis to be done with up-to-date data by using only one data warehouse.

For data to be integrated, it first needs to be transferred to data warehouse from varying sources. The most efficient way to do this is by using direct SQL loads to database. However, direct database connections are becoming less common as cloud computing and in particular Software-As-A-Service (SaaS) software delivery models become more popular. In the SaaS software delivery model, the service provider takes full responsibility for the software provided and thus possibly limits users access to data to REST API interfaces instead of direct database access.

1.1 Purpose of the thesis

This study aims to identify the key advantages and limitations of data warehousing via REST API interfaces compared to the more common SQL based data warehousing. The idea for the Thesis came from an open question from Aveso Oy: what if a data warehouse was implemented using only the available REST API interface instead of direct SQL loads to the database? As earlier data warehouse implementations have been done to large extent or entirely by using SQL loads.

1.2 Research questions

For the Master's Thesis to achieve its purpose three more specific research questions were created together with Aveso Oy. The research questions were designed taking into account the current need for new ways of data warehousing. More precisely, the research questions the Master's Thesis aims to answer are:

RQ1: How SaaS source systems change data warehouse implementations?

As SaaS source systems grow in popularity, they are likely to become the most common source system for transferring data to data warehouses in the future. Yet the current data warehousing process is based on principles that were designed before SaaS source systems became mainstream. Before this due to direct access to database the loads have remained technologically more or less the same regardless of the source system. This Thesis tries to answer the question of whether or not new technologies in source systems will change data warehouse implementations.

RQ2: How data loads can be implemented in SaaS source systems?

Currently most of data warehouse implementations are from SQL to SQL, which means that the data warehouse engineer always has an direct access to the source systems database. However, giving direct access to SaaS systems database is not possible or appropriate for all SaaS providers. Providing an API that has access to the data that needs to be warehoused is more common for the SaaS providers nowadays. The second goal of this Thesis is to find out if a REST API based solution for data warehouse is sufficient.

RQ3: What is the impact of REST interfaces on the maintenance of data warehouses?

With the adoption of REST API interfaces, data availability should improve, making it at least in theory easier to integrate source systems to data warehouse. On the other hand, the use of REST APIs raises the level of abstraction and integration is no longer managed at same abstraction level as before. This can lead to performance and possible other problems, as the REST API might not be available for configuration, which may be necessary if it does not meet performance requirements. The last question this Thesis aims to answer is wether or not the data warehouses maintenance is affected by the added abstraction level to data loads.

1.3 Thesis structure

The Thesis is structured in five chapters. The Chapter 1 is the introduction to the Thesis. It gives a general picture of the problem to the reader, states the research questions, and research methods used. Next, Chapter 2 provides the technical background at the appropriate level for the reader to understand the later chapters. The

main concepts in the technical background that will be discussed in this thesis are cloud computing, data warehousing, and application programming interfaces.

Chapter 3 presents the source system used in the implementation and the implementations of the thesis. There are three implementations in total and their purpose is to test different features of the REST API provided by the developres of the source system in order to answer the research questions. Chapter 4 evaluates the results of the three implementations presented in Chapter 3. Finally, Chapter 5 summarizes the Thesis, answers the research questions, outlines the current limitations and suggests what future research should be done on the topic.

2 Technical background

The goal of this chapter is to give a adequate understanding of the various technologies related to the implementation presented in chapter 3. The chapter is divided to three main subsections. Subsection 2.1 presents cloud computing and its different service and deployment models, subsection 2.2 presents data warehousing and common tools and modeling techniques for data warehousing and subsection 2.3 presents application programming interfaces and some of its subsets.

2.1 Cloud computing

Cloud computing is the usage of computational resources, such as services, hardware, or development platforms, where computers owned by the service provider perform the computing. The amount of resources expected from the service provider is defined in a contract between the customer and the service provider, called a service level agreement (SLA)[5]. The purpose of a cloud service is to scale up and down according to the needs of the moment. In such cases, when the customer demands a lot of resources, it is up to the service provider to allocate enough resources to the customer [6]. The resource allocation should be done not only quickly but also without any interaction with the service provider.

The first steps towards the adoption of cloud computing were taken in the 1950s when virtual machines were first hosted on mainframe computers [5]. Later in the early 1960s, John McCarthy presented his idea at the Massachusetts Institute of

Technology to sell computing resources as an on-demand service, now known as cloud computing. His idea was that computers could one day become a public utility, like the telephone network. In his opinion, this new public utility created by computers would become an important part of a new industry. Gartner's latest estimate [7] puts total public cloud sales at over $670 billion by 2024, which makes it safe to say that John McCarthy was right.

Virtualization is one of the main technologies making cloud computing possible [8]. In traditional computing systems, only a single operating system is run on a single physical system. Virtual machines allow multiple operating systems to run on a single physical machine. On a physical machine, the hypervisor is used to manage and control all virtual machines installed on the physical machine. This made it possible to maximize the usage of computing power on mainframe computers, which often were underused. The hypervisor also makes it easier to manage multiple virtual machines, but it is also a single point of failure, as it can cause a lot of problems if it does not work properly [9]. The function of the hypervisor and the difference between the two models is shown in Figure 2.1.

The first implementation of cloud computing was made by the software company Salesforce in the late 1990s when it released its software as a Software as a Service (SaaS) [5]. Other early adopters of this cloud computing were Microsoft with its Platform-as-a-service (PaaS) Azure and Amazon with its Elastic Compute Cloud (EC2) infrastructure-as-a-service (IaaS). However, among the biggest drivers of cloud computing around the 2010s were Google and IBM, which partnered with universities to use university students to research cloud computing [10].

The first Salesforce SaaS release did not, however, make SaaS mainstream. There could be many reasons for this, but in general, the use, speed, and availability of the internet were not at a reliable stage [11]. SaaS applications only started to become more common with the introduction of Web 2.0, after 2005. Around this time,

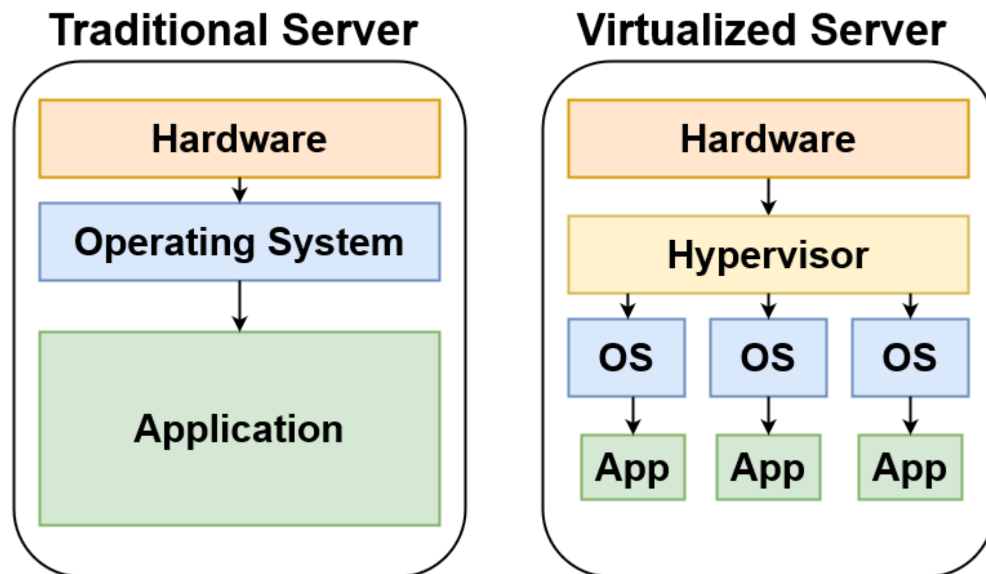

Figure 2.1: The difference between a traditional server and a virtualized server, showing the impact of the hypervisor.

the internet was also increasingly used in the business world. Among the industry drivers of the 2010s is OpenStack, a joint project between NASA and Rackspace, an open source cloud application used by thousands of companies [10]. Also, other popular cloud services were created in the early 2010s, such as Microsoft's Azure. Around the same time, a white paper that explained how cloud service providers can develop cloud services was published by the Cloud Security Alliance (CSA).

Service models

The most common service models for cloud computing are SaaS, IaaS, and PaaS, as described earlier. All models have their typical features, but the common factor is that the service provider controls a varying proportion of the service, depending on the service model.

IaaS refers to the outsourcing of the hardware infrastructure needed for computer computing. Most commonly, this includes only the services listed in Table 2.1, i.e. virtualization, servers, storage, and networks maintained by the service provider. The user accesses the service by specifying the environment requirements for the IaaS service provider [10]. The service provider then configures a virtual machine that meets the user's needs and that the user can access over the Internet [5]. This way the user can, for example, deploy their application on a virtual machine, thus removing the need to worry about their own servers and the associated maintenance costs. On the other hand, the user also loses control of the infrastructure and may not be able to make the necessary changes [9]. Common necessary changes include configuring the firewall, which, depending on the service provider, is not always possible.

In the PaaS model, the service provider provides the user with an environment suitable for software development. In contrast to IaaS, PaaS includes the same services and O/S, middleware, and runtime [10]. The full list of generally included services can be seen in Table 2.1. This allows the user to focus on software development without having to configure their own environment, including hardware and the installation of programming languages and the necessary libraries [5]. Other benefits include automatic scaling of resources depending on the needs of the moment. Similar to IaaS, the user loses control of the services managed by the provider but manages applications deployed on the PaaS platform [9].

In SaaS, the service provider offers the entire software over the internet via a browser interface, without the user having to install anything on their own compute. As shown in Table 2.1, the service provider manages the same services as in PaaS, as well as data and applications. So, contrary to PaaS, the user does not manage applications and cannot deploy their own applications in the SaaS [10]. With the loss of almost all control of the application, there is not much the user can modify.

However, applications often allow for customization, such as interface refinement or the creation of custom functions, etc [5]. But in some SaaS applications, such as Netflix or Office365, the need for customizations is not as relevant. The most relevant for this work are ERPs provided as SaaS, where customization can be an essential part of the overall solution [6].

	IaaS	PaaS	SaaS
Applications			x
Data			x
Runtime		x	x
O/S		x	x
Virtualization	x	x	x
Servers	x	x	x
Storage	x	x	x
Networking	x	x	x

Table 2.1: A table showing what different services are managed by different cloud computing service models. [10]

Benefits and challenges

Arguably the main reason organizations adopt cloud computing services is the reduction of cost in IT services [9]. For most organizations, there is no reason to handle all the necessary IT in-house, as this can lead to the need for an internal IT department. There are costs associated with buying IT systems in-house, updating them over time, maintenance, etc. Cloud computing services are simply billed by storage used or by time used. Another important reason is the flexibility of the services [5]. Services scale with the number of end-users and also with the potentially increasing performance requirements of the customer's software. And since cloud computing is the main business of the cloud service provider, the service is likely to be more reliable than in-house server setups.

Some of the biggest challenges raised by cloud computing relate to the confidentiality, integrity, and availability (CIA) triad. Confidentiality means that only users that are authorized with permissions can access certain information, integrity means that the accessed information needs to be correct, and availability means that the information is always accessible to authorized users with permissions. While these are the basics of cyber security, their implementation in the cloud cannot be taken for granted. Managing access to shared resources is difficult and can cause confidentiality issues, changes that break integrity can be difficult to detect, and availability can be lost due to things like provider outages or internet connectivity problems. [12]

One of the main data-related problems is data lock-in, where once data has been given to a service provider, it can be difficult or impossible to transfer it to another service provider [5]. For example, the reason for switching to another provider may be technical, such as better performance, or the same service at a lower cost. In such cases, the data is entirely the responsibility of the cloud service provider, which can be too much of a risk for many companies, as switching to another service provider could simply be not possible. The problem could be solved by creating data standards for cloud services, removing the fear of data lock-in and allowing data-critical companies to use cloud services with greater confidence [12]. Lack of such standards may lead to significant business damage if a company chooses an unsuitable cloud service provider

In the scope of this thesis, the main problems are related to data and its availability. As in SaaS source systems the data is handled by the service provider, the user may not know where and how the data is stored. This is partly the idea behind the use of cloud computing as service provides takes responsibility of it. In some situations however, a user may need additional information about their data, for example, when purchasing data-related services from a third party.

Deployment models

Deployment models in SaaS and other cloud computing services simply control who has access to the service and how it is accessed. There are four different deployment models: public cloud, private cloud, community cloud, and hybrid cloud. Of these, the public and private clouds can be thought of as opposites, while the community and hybrid clouds combine different aspects of these.

In the public cloud, deployment is available to anyone through the internet. Public clouds are designed to serve large numbers of customers, which may be individual or corporate [5]. Inside the cloud, every user is using their own virtual environment separate from other users' virtual environments to allow monitoring and high performance. The private cloud, on the other hand, is only used within an organization, although the principles are the same as for the public cloud [10]. The private cloud is primarily designed to improve the security of company data by allowing users to be restricted by controlling the firewall.

A community cloud is a cloud model shared by two or more organizations. It combines the public cloud multi-user principle with private cloud user control [10]. Often, organizations using the community cloud have common goals or other reasons why it serves them better than other deployment models. A hybrid cloud can be referred to as anything that combines multiple deployment models into one [9]. Most commonly, this means that an organization has some confidential services or data that needs to be run or stored in the private cloud, while some services are accessed through the public cloud. The hybrid deployment model allows organizations to maintain a public cloud for non-sensitive data and a private cloud for critical data [12]. Because of these characteristics, most organizations prefer the hybrid deployment model to others.

Cloud ERP

Enterprise Resource Planning (ERP) systems are software used to manage a company's different operations from a single central software solution. They became more common in the 1990s and have been very popular in various sectors, especially among small and medium-sized enterprises [13]. One factor explaining the popularity of ERP systems is the ever-increasing data needs for product manufacturing and decision-making [8]. ERP systems are able to manage data flows more easily, which is why they are now almost an essential part of business operations.

There are two main ways to implement cloud ERPs: ERP on IaaS and ERP as SaaS [6]. In addition to these, there are the so-called on-premise ERPs that are entirely deployed on-site, without the usage of cloud services. Each has its own strengths and weaknesses, and one is not necessarily worse than the other.

In traditional ERP systems using middle-tier architecture, the middle-tier including the web server and applications server, and the data tier including the database server are managed by the organization [6]. Managing own data tier is a crucial factor in the sense of this thesis as in traditional implementations data availability is high, compared to SaaS implementation where data availability is low. An example of middle-tier architecture can be seen on Figure 2.2. As a result of managing ERP inside the organization, IT skills are needed within the organization, as servers, data, the ERP system, etc. need to be managed. This results in large direct and indirect investments in IT, which may also lead to escalating costs for the company [8].

The lower cost of ownership, upfront, and maintenance of cloud-based ERP systems is one of the main reasons for the change [8]. It is also much easier to deploy cloud ERP than on-premise ERP, which is particularly important for companies purchasing their first ERP. The usability of cloud ERP also influences the adoption, because regular ERP needs to be installed on a computer, while cloud ERP can be

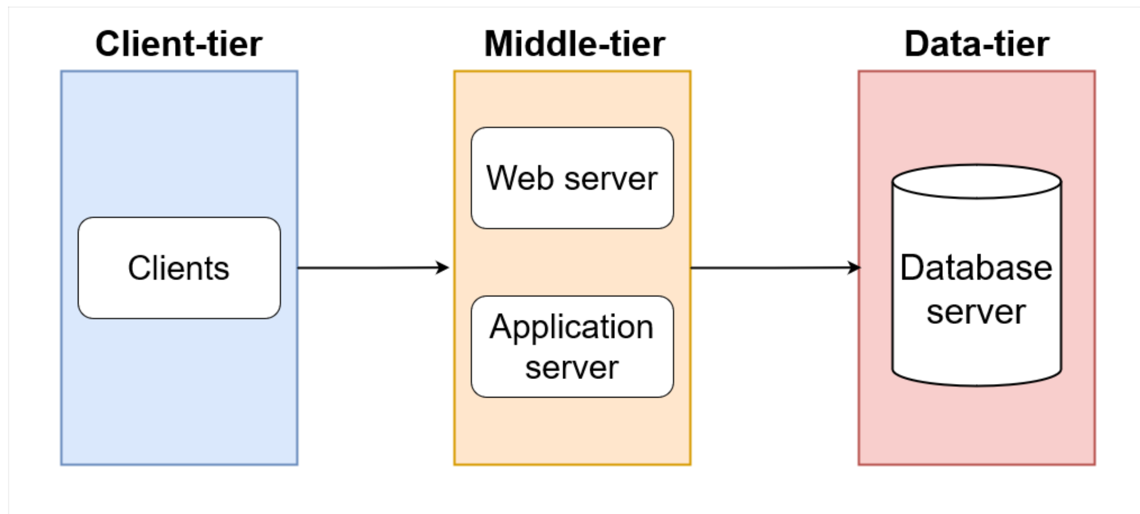

Figure 2.2: Common middle-tier architecture [6]

used in a browser [13]. Cloud ERPs can also offer better performance, upgrading, and support than traditional ERPs. However, these depend more on the service provider, and it is by no means clear that this will always be the case.

In ERP on IaaS service provider provides the infrastructure where the ERP is going to be deployed. This means virtualization, servers, storage, and networking, as seen in the Table 2.1. The service purchaser is responsible for choosing the appropriate ERP for the situation and required licenses. Compared to ERP on SaaS this way service purchasers are able to tailor their ERP systems more extensively, but the costs are often higher. [6]

In ERP as SaaS service provider manages all the services needed for ERP deployment. It includes all the same services shown in Table 2.1. SaaS ERPs are a common way of using ERP services nowadays. Despite its simplicity, one of SaaS ERP's problems is data safety [6]. As the whole SaaS service is managed by the service provider, the SaaS service provider has access to all the organization's data which can lead to security issues. Another weakness of Saas ERPs is that tailoring the ERP to better fit the industry it is used in is expensive [8]. For this reason, large companies that need a highly tailored ERP are hesitant to give up their on-premise

ERP solutions. Fully cloud-based ERP, on the contrary, is well suited for small and medium-sized businesses that require little, if any, customization.

SaaS ERPs suffer from the same problems as traditional SaaS services. Major problems, as in regular SaaS, are related to data privacy, cloud transparency, and data security. For example companies may need the data to be located in a limited geographical area because of different regulations, which the service provider may not be able to [8]. This is emphasised because business data often requires more sophisticated processing than that of individuals, but when using SaaS, the location of the data may not always be clear. There are also some technical problems associated with moving the ERP system to the cloud, as once the ERP system is in the cloud, it can be difficult to integrate it with other IT applications [13]. Integrating other IT applications with ERP systems can already be difficult, but the introduction of cloud computing may make it even more difficult.

2.2 Data warehousing

Data warehouse is as a read-only database that handles data from a different operational sources for end-users. Data warehousing first emerged in the late 1980s and it has remained relevant ever since. Some of the key features of data warehousing is saving the time information associated with the data, keeping the data unchanged, integrating the data, and using the saved data in business operations [3]. Some other typical features of a data warehouse include storing large amounts of historical data, managing multiple operational data sources simultaneously, and managing changes in data sources [2]. How well a data warehouse can manage these is influenced by the architecture chosen and the data model used [4].

Business Intelligence (BI) is the process of using computing to analyze data collected from a company's processes [3]. One of the most important components of the business intelligence process is a stable data storage that can be easily integrated

with different data sources, for which the data warehouse is used. The BI process turns large amounts of raw data into information that helps the company's decision making [4]. This is essential for any company that wants to stay competitive in today's market.

One popular alternative for data warehousing is data virtualization. Data virtualization, like data warehousing, supports the use of multiple sources and near real-time data collecting. Data virtualization creates a so-called data virtualization layer, which combines different sources and creates a single virtual view of them. From the virtual view, data can be retrieved as if the virtual view were actually one coherent view. This can be cheaper and simpler compared to data warehousing. However, virtualized layers may also lead to performance degradations, which is why data virtualization is out of the scope of this study. From now on, this Thesis will focus exclusively on data warehousing. [3]

Data Warehouse Modeling

The data warehouse model defines the technical architecture used for the data warehouse. Using the right model for the specific situation is important, as the model influences how the data warehouse handles data and manages risk. In total, there are three main ways of implementing a data warehouse: Inmon's approach from the '90s, Kimball's approach from the '90s as well, and a more recent data vault approach from the 2000s [2].

Inmon's approach is a data warehouse model based on entity-relationship modeling [2]. Its architecture is based on four levels: first level is operational which loads all the source data, second level is a data warehouse with its ETL operations, third level which includes data marts, and lastly end users and tools that access the data from data marts. Data warehouses and data marts should be physically separate from each other as they serve different purposes. Data warehouses are designed to

store as much data as possible with its history, while data marts serve the changing needs of the end user. By physically separating them, you can be more confident that both components will perform to their performance requirements. Since data marts pull data from the data warehouse, changes are first loaded into the data warehouse and then propagated to the data marts used by end users. [1]. This is why Inmon's approach is called the top-down method in data warehouse modeling.

Kimball's approach was created afterward Inmon's approach as an alternative way of data warehouse modeling. One of the fundamental changes in Inmon's approach was that Kimball's approach was based on dimensional modeling instead of entity-relationship modeling [2]. In practice, this meant that when in Inmon's approach data loads were done regardless of end-user needs, Kimball's approach was based on the strong involvement of end-users throughout the whole process. Another fundamental change is in the architecture, where there is only the ETL process on the second level, and on the third level is the data warehouse as a collection of data marts accessible to end users in the last level. Because the data marts form the data warehouse, Kimball's approach is called the bottom-up method in data warehouse modeling [1]. In general, the top-down method is better at handling change management, whereas the bottom-up method handles reporting and analytics better.

For the last couple of decades a third approach, called the data vault approach, has been used for data warehouse modeling. It was designed by Dan Linstedt. The popularity of the data vault method is well illustrated by the fact that Bill Inmon, who developed Inmon's approach, proposed the data vault method as a new generation of data warehousing modeling. [4]

The data vault approach is a fundamentally different process designed to fix possible problems that could occur as the quantity of data grows with Kimball's and Inmon's approaches. For example, the Inmon and Kimball models may require

laborious changes if the source data or analysis needs to be modified. Information separation is the most fundamental change compared to Inmon's and Kimball's models. Also, as seen in Figure 2.3 the architecture is changed from four levels to five levels. The first level remains the same, the second level is a staging area, that contains a database that holds raw data and possible additional metadata loaded from the first level, the third level is the data warehouse, where the raw data is loaded, the fourth level contains of data marts that are accessed by individuals, which form the last level. [2]

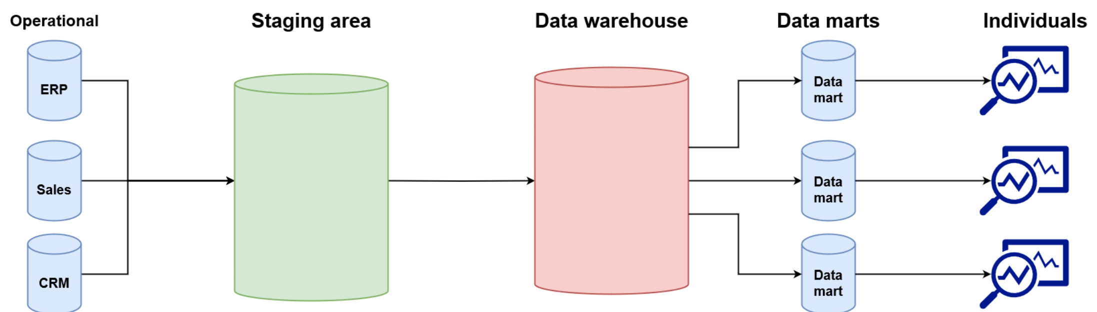

Figure 2.3: Common data warehouse architecture used with data vault modeling

Data vault

When first created, one of data vaults goals was to create an empirical approach to data warehousing [4]. Dan Lindstedt tried to achieve this goal by dividing objects in two: an entity with a unique identifier, and the attributes of the entity. The idea for this split came from the fact that objects in business systems usually contain a single unique identifier, while its other attributes may change. The data vault approach separates information into three entities: hubs, links, and satellites [2]. This split of information avoids creating changes in the data warehousing environment that

could break the data model. In addition to this, some other features of data vaulting are keeping all the data intact and allowing for multiple parallel loads at once.

Hubs represent some real-world entities, such as a customer or a product. Hubs are always uniquely identified, such as through a customer ID or product ID [2]. The unique identifier, whatever it may be, is designed to either never or just very rarely change [4]. This can be seen either as an advantage or a disadvantage of data vault. For example, it is not suitable in situations where objects do not have a unique key.

Satellites are used to store the attributes of the hub and metadata [4]. Satellites can also be attached to links and there can be more than one satellite for each hub or link. If there are multiple satellites they can be broken down according to different requirements, such as what kind of information they contain or how often the satellite data changes. Each satellite can be connected to only one hub or link [2].

Links are used to connect hubs to other hubs [2]. Links cannot contain any attributes, as if there were they would be saved to a satellite that is connected to the link. In a rare case, the link can also be connected to another link [4].

Figure 2.4. illustrates a simple example of how a data vault would create hub-link-satellite entities from customer and customer order tables. The customer table is split into customer details and contacts which, according to their name, store different customer-related attributes. In this example, the customer order table only creates one satellite to store all the attributes from the customer order table, which is why the name of the satellite does not contain any information about the satellite. The two hubs are connected by the customer order link, which contains both of the hubs' primary keys, as foreign keys and has its own primary key. This is a simple way of connecting two hubs. Also, all the entities in this example contain some metadata, which in this case is load date, describing when the data has been downloaded, and record source, describing where the data is from.

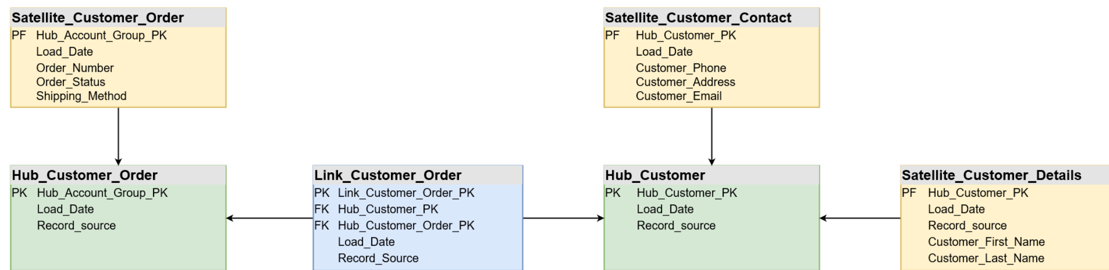

Figure 2.4: An example of data vault architecture describing different components in data vault of customer and customer order tables

ETL and ELT

Creating a data warehouse requires retrieving data from source systems into the data warehouse. However, the data queried from source systems can, for example, be of poor quality or be in different format, so the data needs to be transformed between the two systems [1]. For this, an extract, transform, load (ETL) process is used. ETL process can be seen as an enabler of data warehousing because it is a critical part of the overall data warehousing solution. According to some studies, 60-80% of the total data warehousing work is spent on the ETL process alone [14].

ETL consists of three stages [1]. Initially, all relevant data is extracted from one or more source systems for transformation. The transformation phase involves the application of various data modification rules, such as the elimination of duplicate data, to ensure compatibility with the data warehouse. Once the data has been transformed it will be loaded into the designated data warehouse. The progression of the steps is illustrated in Figure 2.5. All the activities involved in the ETL process must work together to form a solid and properly functioning ETL workflow. During the design of the ETL workflow, several factors, including performance and the integration and availability of source systems, must be taken into consideration [14]. Thus, the ETL process involves programmers and architects.

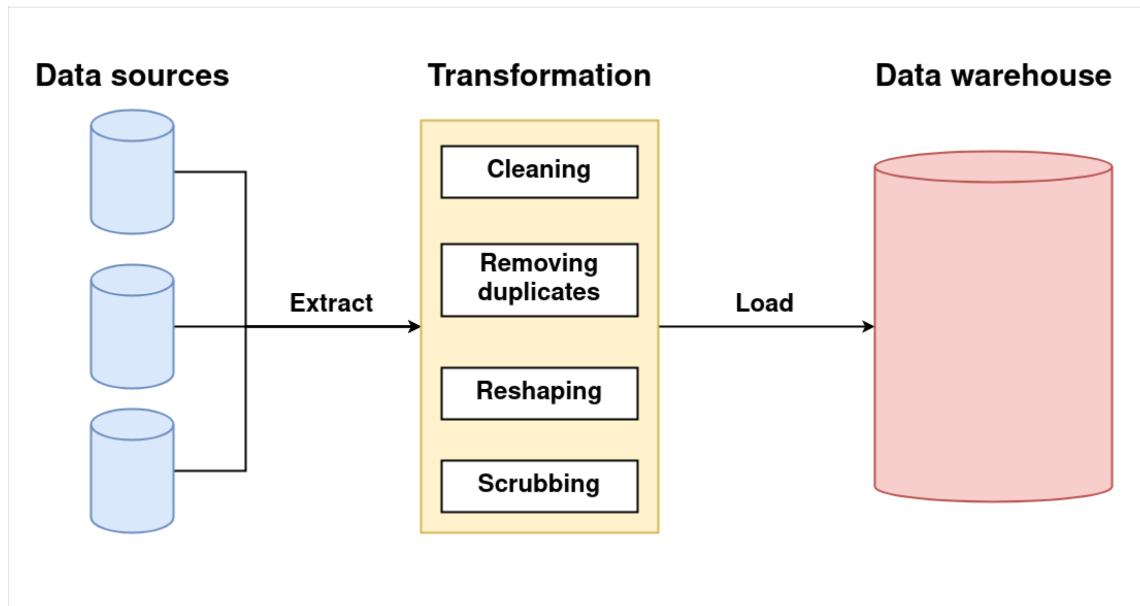

Figure 2.5: A figure explaining steps of a extract, tranform, and load (ETL) process

A recent ETL challenger is a similar process called extract, load, transform (ELT). It is otherwise similar, but as the name suggests, the loading into the data warehouse is done before the transformation. One of the reasons for preferring ELT to ETL is the single source of truth approach [4]. Since the data of source systems is exported unmodified to a data warehouse, it captures all the source data from the source systems, unlike ETL. According to [1] ELT can also be more scalable, perform better, and handle large source systems better, but because it is a newer process than ETL it simply lacks tools and expertise.

2.3 Application programming interfaces

Application programming interface (API) is a general way of integrating computer systems. Computer applications use APIs to gain programmatic access to the data of another computer application or, more precisely, the access data from the database to which this application is linked. API development is one of the key enablers of

the migration of monolithic software to the cloud and is therefore at the heart of the design of today's digital systems [15]. In information systems science, APIs are studied as a new boundary resource, meaning a resource between an owner and a third party [16]. Other boundary resources include technical documentation and educational materials, although APIs are fundamentally different from these.

With the growing popularity of APIs, the API economy, which refers to the creation of value for businesses through APIs, has gained more attention [15]. One of the biggest value-adds to the API economy is Web APIs, which provide access to the API backend via the internet. Web API development and management is not simple and often requires involvement from several different stakeholders [16]. The key stakeholders in API development are:

- the backend provider, who is responsible for the functionality of making data available to the API
- the API provider, who is responsible for the functionality, documentation, and availability of the API to the consumer
- the API consumer, which in this context refers to the application that is used by the end user

APIs are also often associated with other stakeholders, such as sales, support, and legal.

APIs are difficult to design, and it is difficult to create guidelines for their design. There have been many attempts, but few generalized methods. One example is Bondel et al [16] design science-based method to create a framework for API design that brings different stakeholders to the API management team to support their work. The key idea of the framework was to use three different cycles to support and guide the design and collaboration problems. The different cycles are:

- relevance, which ensures that an API solves a real business problem

- rigor, which ensures that existing knowledge is applied in the best possible way
- Design, which is the phase in which the actual API is built using information gathered in earlier cycles and then evaluated to allow for changes

. With the help of the cycles, a pattern language was made that provided a top-level solution to potential problems. For example, not always both rigor and relevance cycle is needed after an evaluation. In case there is a need for more knowledge, only a rigor cycle could be run and after it, the API could be re-designed in a design cycle, and evaluated again. Or in a case, where the API design has drifted away from the business problem, only a relevance cycle could be run, and so on.

REST API

Web APIs can be considered as a subset of APIs. What all web interfaces have in common is that they open their endpoints to the internet, i.e. they can be accessed via the internet and the APIs are accessed using the Hypertext Transfer Protocol (HTTP) [16]. This means that one can assume that at least Post, Get, Put, and Delete operations are supported by the API. Within the subset there are several different popular solutions to implement web APIs such as SOAP, REST, gRPC, and GraphQL [15]. Of these options, we only consider REST because the source system only provides REST APIs, thus the implementations are dependent on REST APIs.

REST was invented more than twenty years ago as a design guideline for REST APIs and other services and has since grown in popularity. Despite its age and popularity, the design and use of REST APIs is still an active discipline [17]. The small differences in each REST API design create a situation where the API user has to independently explore the different functionalities of the API. On the other

hand, it is precisely this ideology of intervention in every activity that can be seen to have led to the great popularity of REST APIs [18]. The idea of a REST API in simplified format is illustrated in Figure 2.6.

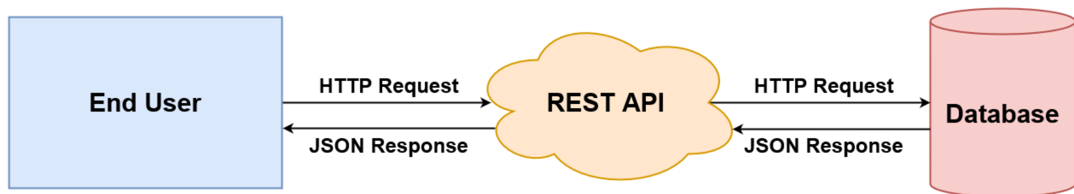

Figure 2.6: An example of how REST API returns data from database to end users

REST is an abbreviation for Representational State Transfer which is an architectural style of software development that is intended to guide the design and development of applications for the Web [18]. Some of REST's principles are resource addressability which means creating unique Uniform Resource Identifiers (URIs) for all resources, and resource representations which means that the internal data can be in any form, but the end user handles some representation of it, typically JSON or XML, uniform interface which refers to each method being accessed the same way and by using HTTP, and lastly, statelessness which means that the server does not have an interaction state, but all the information is in requests between client and API. Although these principles are at least theoretically true for the REST API, difficulties have been identified in the literature in areas such as statelessness, uniformity, and representation [17].

API Management

APIs can be divided into two main categories: open and closed [15]. An open API can be used by anyone, while a closed API can only be used by a specific group of people who have access to it. Closed APIs are used, for example, within

organizations, when an API is used to process information that does not belong to people outside the organization. The third category into which APIs are sometimes classified in the literature is the group category [16]. Group category APIs are closed, but with the required permissions they can also be used by people outside the organization. This type of APIs are common in a business-to-business situations, but they increase the complexity of the API, possibly making it more exposed to vulnerabilities.

APIs make it possible to create secure connections between different systems. Properly categorized APIs can be used to create secure and monitored connections. On the other hand poorly defined API permissions are one of the biggest security risks of an API [15]. Authentication is often done using third-party services such as Oauth, which requires expertise to maintain and configure. However, Oauth is only one option among many other authentication solutions, so choosing the best one is not a simple choice. The literature identified the need not only for more explicit authentication but also for more straightforward testing of API security [17].

The problem of standardizing APIs is not just limited to authentication but actually applies to the whole design of APIs [17]. Differing use of HTTP verbs, different documentation policies, and poor error messages are all examples of the lack of standardization. The problem with API standards, however, is that there is no clear authority that should be in charge of creating the API standards. While there is no general consensus on how APIs should be provided, EU's view is to favor the use of open, standardized, and well-documented APIs, which are seen to increase innovation in the future [15].

3 Integrating SaaS source systems to DW: IFS case study

Based on the literature review, a minimum viable product (MVP) was developed. The product is designed around the key components presented in Chapter 2. The product uses a cloud-based ERP system as the source system for all the data used. The integrations use the REST APIs of the ERP system to create a data warehouse that implements the ETL approach. The product is presented in more detail later in this chapter, along with the test implementations.

3.1 IFS

Industrial and Financial Systems AB (IFS) is a Swedish software company founded in 1983. IFS has more than 500 partners and serves thousands of customers worldwide [19]. IFS is a recognised provider of a variety of enterprise software, with ERP, Enterprise Asset Management (EAM) and Field Service Management (FSM) systems as its main areas of expertise. In the year 2024 IFS had a total revenue of more than 1.2 billion euros of which more than one billion was part of annual recurring revenue [20].

The main product of IFS is their platform called IFS Cloud. IFS Cloud is a combination of ERP, ESM and FSM modules tailored to the needs of the user [20]. The strength of IFS Cloud is that it offers targeted solutions for specific

industries [21]. Such industries include, but are not limited to energy, aerospace and manufacturing.

3.1.1 IFS API

This section provides a more detailed description of the IFS API. It includes an introduction to Odata, the standard behind the API, a brief description of the API architecture and the IFS-provided official way to use the API.

Odata

Open Data Protocol (OData) is an ISO/IEC approved technical protocol for REST data querying, on which the IFS API is based. OData supports HTTP protocols and methods, such as GET, POST, PUT, PATCH, and DELETE to access the API. This allows users to perform data queries using efficient URL-based syntax. It also supports filtering functionality that allows users create queries for only the data they need [22].

IFS business entities are exposed as OData API endpoints. These API endpoints follow an IFS-specific syntax, which looks like this:

```
http://localhost:8080/main/ifsapplications/Customers.svc/customerInfoSet?$filter=Id gt 1000
```

In the example URL localhost is the server, 8080 is the port used to access the server, main is the context used in the API load and ifsapplications is a constant. Together, these four values form the Servlet Path. Customers.svc is the name of the service that will be used in this particular data load. Services are more commonly called projections in IFS and, in simplified terms, projections are a superset of entities. The customerInfoSet is the resource that is accessed within the service. It is one of many possible entities accessed within the projection. Last in the URL is a filter that filters all IDs that are less than 1000 out of the response. Filter is also

called a query option and it together with the accessed entity is called an OData access [23].

Requests sent to OData API endpoints return HTTP status code and response body to the user. From HTTP status codes the end user will know whether or not the request was successfully handled. For example, if status codes in range 200-299 are successful, status codes in range 400-499 are client errors and status codes in range 500-599 are server errors. In successful queries response body contains the queried data in JSON format, in errors it contains more information on the error [23].

IFS Architecture

At a general level, the architecture consists of three compulsory servers. The database server is an Oracle database instance, the middle tier server works as the application server running IFS Cloud, and The management server is used to maintain the deployment, containing all setup and installation information, updates, and configuring scripts for middle-tier server. The servers are either deployed as a full SaaS offering managed by IFS, called a cloud deployment, or deployed in a customer-owned environment, called a remote deployment. In the implementations of this Thesis, the latter is used. The overall focus will be more on what happens on the middletier and database servers [24].

Middle-tier servers main technical component is a Kubernetes cluster used to manage small web applications, called services. These are running on docker containers. As seen on the Figure 4.1, end-users connect to middle-tier server services in various ways, such as through the browser, mobile apps or integrations, via REST APIs. As explained earlier, the requests are likely to be simple HTTP requests, such as get some data to display on the screen by IFS Cloud user interface. For example, when an end-user doing integrations queries to get customer order data

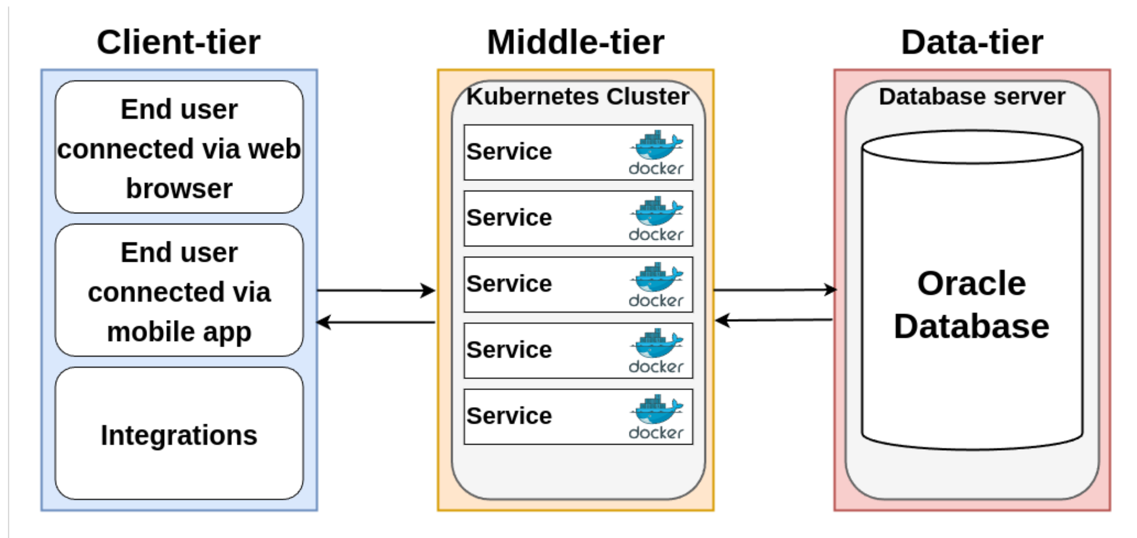

Figure 3.1: IFS Cloud middle-tier server architecture presented in the same way as in literature review, Figure 2.2

from database, it sends the requests to a specific REST API URL which returns this data. More specifically, the request would go through the Kubernetes cluster on the middle-tier server, where it would be directed to the OData pod which is responsible for the REST APIs. From there the data would be queried according to the parameters in the REST API URL, and then returned to the end-user.

The database and middle-tier together form the so called core application services of IFS Cloud. Database holds the data and the business logic, while middle-tier is responsible of the APIs used to access the business logic [24]. Middle-tier hosts own pods for such functions as identity access management, reporting, and mobile service [25]. Arguably the most business critical pod in general is the OData pod that hosts the OData service, as all REST APIs are dependent on it.

OData service is the main architectural component for REST APIs on the middle-tier server. It handles a large part of both integration and client logic, as both are highly dependant on REST API requests in IFS Cloud [26]. For integrations requests to OData are simply transformed from HTTPS request to JSON responses by OData service, allowing the user to create queries via REST API to

Figure 3.2: IFS API Explorer UI

database. For client the same HTTP protocols and REST APIs are used, but the main difference is that the responses are ran through an code engine that visualizes the JSON data in different ways instead of returning raw JSON responses.

API Explorer

API Explorer is the main way to access the IFS APIs from IFS Cloud user interface [27]. As illustrated in Figure 3.2, the API explorer offers a user-friendly user interface for accessing the IFS APIs. It provides basic details such as name, description, documentation, and specifications for different versions of OpenAPI and OData for all APIs available.

API Explorer categorizes APIs according to three main attributes: API class, layer, and deprecated [27]. The API class divides APIs to standard APIs and premium APIs, which are otherwise similar, but premium APIs are designed to solve specific business problems and have more extensive documentation and a long-term support. The layer attribute defines which projection the API is part of core, customization, or configuration. The core is all standard REST APIs provided by IFS,

the customization REST APIs are done either by modifying or creating new source code in IFS Cloud, and the configuration REST APIs are those that depend on minor changes to IFS Cloud that do not affect the source code. Deprecated is either false or true, indicating whether or not use of the API will be supported in the future.

3.2 Case study

To better answer the research questions, three implementations were created to test different realistic situations in a test environment. The first implementation 3.2.1. explores if performing a larger data load from a single source table from source database to target database is possible by using the REST API. Second implementation 3.2.2. explores how well IFS REST API handles IFS custom objects. The third and final implementation 3.2.3. demonstrates how multiple table data warehousing can be achieved by only using different REST APIs. To answer the research questions, these implementations will be analysed as a whole and separately for their pros and cons.

The technical process used in the implementations follows the usual structure of the ETL process, represented by Haryono et Al [1]. There will be only one source system used, which is an IFS Cloud ERP system and from there the data is extracted using only the REST APIs provided by IFS. ETL part of the data warehousing process is done in Azure's data factory which is a popular data integration tool developed by Microsoft. Data factory handles the GET request to IFS REST API, handles the transformation of the data from JSON to SQL, and exports the data to the SQL server. Azure SQL server is used as a data warehouse, where the data is eventually saved. All main parts of technical architecture are shown in figure 3.3.

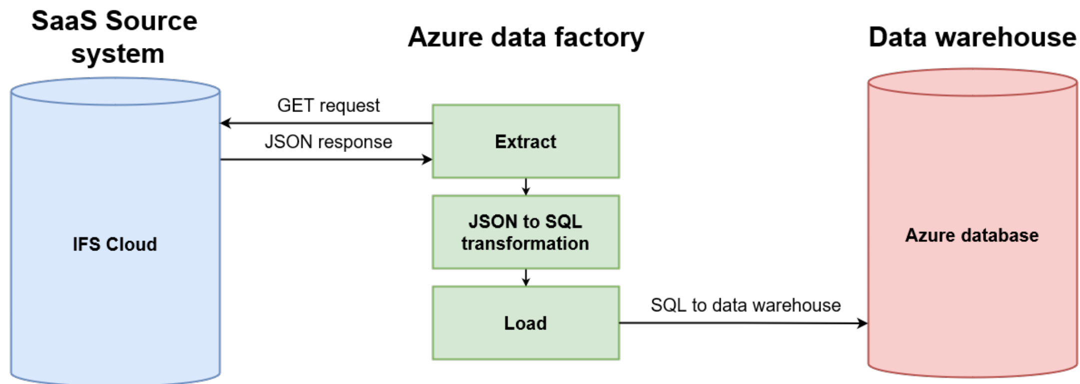

Figure 3.3: Technical architecture used in test environment for implementations

3.2.1 Loading a single large table

The first implementation shows how data transfer via IFS provided REST API work from one table with a large amount of data to a single target table. The implementation is going to test the transfer of data from the same table with same untouched data at different amounts. The test is performed at five different levels. At the first level, 100 000 rows of data is transferred from the source table to the target table. At the following levels, the number of rows to be transferred increases to 250,000, 500,000, 1,000,000 and finally up to 2,000,000. The purpose of this implementation is to analyze whether large data loads would be a bottleneck in IFS REST API data warehousing implementations with larger data loads.

The source data used in this task is simply 100 000 lines of voucher data, copied multiple times to the source table until there was enough data for all the tests. The most important thing about the source data was the number of rows, rather than the data content of the table. The reason for choosing a voucher table for this use is that it is often one of the largest tables in a company's database.

To make it easier to compare the results, a similar implementation with direct SQL to SQL loads from one database to another was also produced with the same

amount of rows. This way it is possible to see how much time difference there is when doing the same data load as traditional SQL to SQL data load and REST API data load and whether the time difference is linear or not between the two implementations. The results of the implementations are compared at the end of the section.

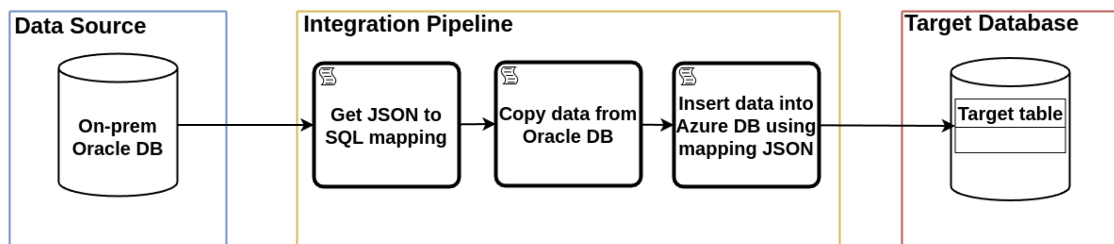

Figure 3.4: Figure shows how loading a single large table was coordinated between databases and Azure data factory pipeline

To achieve the wanted functionality an ETL pipeline, seen on the Figure 3.4, was built on Azure data factory. The pipeline includes a simple copy data and insert data activities which use the REST API to retrieve data from the source database and copy it to the target database. The retrieved data is the content of the voucher table in JSON format. In order for the pipeline to integrate JSON voucher data correctly, it needs JSON-SQL mapping instructions. For this purpose, a database function has been created in the target database that creates a JSON mapping for given table. Data factory first calls the function and then uses its result as the mapping to insert the data retrieved in JSON format from the source database to the target database in SQL format.

The target table could be created automatically in data factory, but then the data types of the columns in the target table would be in NVARCHAR(MAX) format, which in turn increases the size of the table. In this case, the target table is created in advance using a tool that created needed table to Azure database by using Oracle table's schema information as input. Optimizing the size of the table is not in the

scope of this Thesis, but it is one of the problems associated with data warehousing as the speed of data transfer is affected by the size of the data.

The first test where data was retrieved as JSON and written as SQL using IFS REST APIs for data loads resulted in rather long copy times from source table to target table. As seen from the table 3.1. the copy times were as following: 7 minutes for 100,000 rows, 17 minutes for 250,000, 37 minutes for 500,000 rows, 69 minutes for 1,000,000 rows, and 139 minutes for 2,000,000 rows. The results indicate that the time increased roughly linearly as the number of lines increased, as one would expect. Additionally, the transfer time for 100.000 lines stayed around 7 minutes for all test cases for this implementation.

Number of rows	**Load size (MBs)**	**Write size (MBs)**	**Time (minutes)**
2,000,000	4,382	786	139
1,000,000	2,199	393	69
500,000	1,109	196	37
250,000	563	98	17
100,000	226	39	7

Table 3.1: The table shows the number of rows retrieved, the size of the JSON file retrieved as load size, the size of the SQL written as write size, and the time spent retrieving and writing the data as time

Another thing that was of interest in the implementation is the size of the data in different formats. As seen from table 3.1. the read data in the JSON format was roughly 5.5 times the size of the same data in SQL format for all test cases. This was to be expected, and it also shows how much less data transfer in terms of size the data migrator needs to do in cases where there is direct access to the database. The only way to solve this problem would be to replace JSON with a new data format that stores data in almost the same size as SQL or by compressing the retrieved data. However, solutions like this are not commonly used yet. Otherwise, the test indicates similar results as when comparing copy times, as there is a roughly linear increase in both load and write sizes between the different test cases.

Number of rows	Read size (MBs)	Time (minutes)
2,000,000	657	40
1,000,000	327	19
500,000	163	9
250,000	141	4
100,000	31	1

Table 3.2: The table shows the number of rows retrieved, the size of the SQL data retrieved from the source system as load size, and the time spent retrieving and writing the data as time.

The another test where data is retrieved as SQL and written in SQL resulted in much shorter copy times, as expected. Comparing the results in table 3.2. to results in 3.1. one can see that the copy times in JSON to SQL are roughly 3.5 times longer than the copy times in SQL to SQL for the same number of rows. The difference is minimal for the smallest copy of 100,000 lines as it is one minute compared to seven minutes, but the difference is already significant for downloads of 2,000,000 lines becoming over 90 minutes longer. The test proves that SQL to SQL data loads are much more efficient when compared to REST API loads, although evaluating the differences in depth is outside of the primary scope of this Thesis.

Other points worth noting when comparing the two cases is that SQL to SQL copy does not require a separate mapping function, like JSON to SQL copy. This not only saves time, but also removes the bug-prone part of data migration. When comparing Tables 3.1. and 3.2. one can also notice that although it is the same data that has been transferred, the data read in Table 3.2. is always slightly smaller. This is because the source data is in the Oracle database, while the destination table is in the Microsoft SQL server. Although the difference is small, it can be taken into account, especially in data intensive implementations.

3.2.2 Custom attributes in the target table

One more IFS specific concern to solve regarding the REST API loads from SaaS source systems is that how dynamically the REST API handles IFS custom attributes. Custom attributes are an IFS concept that are used to add an attribute to an existing entities [28]. It is crucial for the overall solution that these work in the same way as normal attributes, as it is often necessary to integrate all attributes of an entity into a data warehouse. If the custom attributes would not work in the same way, this would have a negative impact on the maintainability of the solution.

In this implementation we are going to use a view that contains data about inventory parts called INVENTORY_PART_CFV as the data source. In the database the main entry point to inventory parts would be a view called INVENTORY_PART, without the CFV extension. However, the INVENTORY_PART view does not maintain any custom attributes. For custom data, INVENTORY_PART view is connected to a INVENTORY_PART_CFT table that maintains the custom attributes for each record. The views INVENTORY_PART and INVENTORY_PART_CFV and INVENTORY_PART_CFV are all connected by an unique identifiers OBJKEY and ROWKEY as seen in Figure 3.5.

The INVENTORY_PART view is associated with a optional 1:1 relationship to INVENTORY_PART_CFT table. This means that there is one record for every entity in INVENTORY_PART_CFT table, that maintains all custom attributes for a single entity in INVENTORY_PART view. This means that there for every record in INVENTORY_PART view there is a record in INVENTORY_PART_CFT table, if the record has custom attributes. If the record does not have custom attributes, there will not be a record in INVENTORY_PART_CFT.

The record is only created once there will be data in custom attributes of the record, i.e. the custom attribute is created and its value is not null. The relationship can be seen in Figure 3.5, where the vertical line at INVENTORY_PART

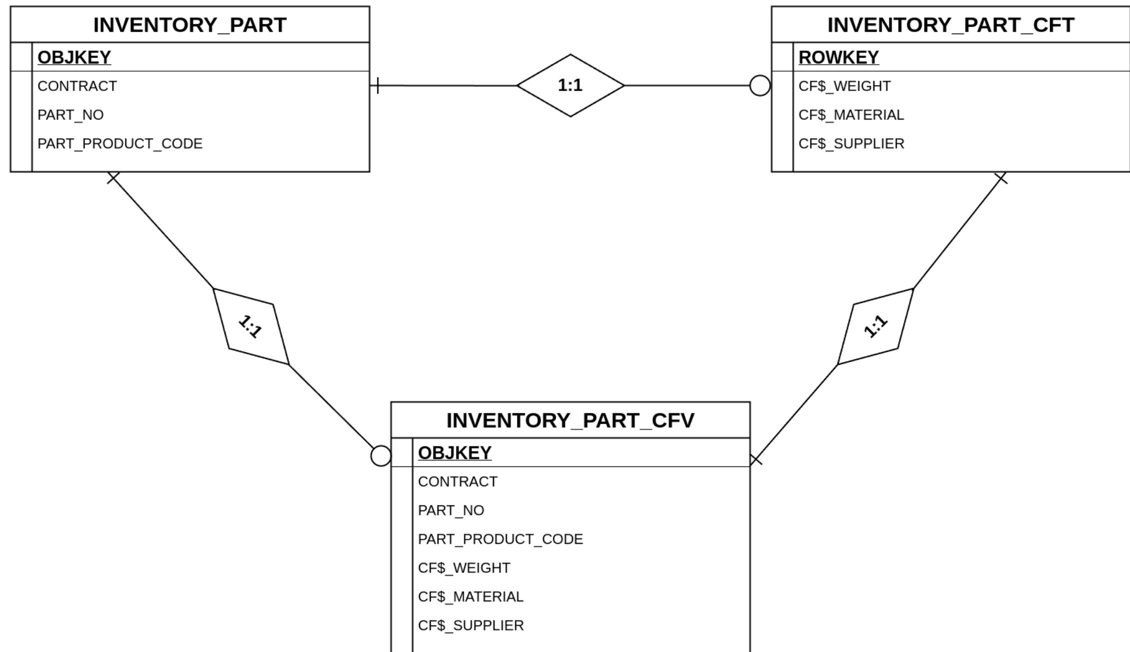

Figure 3.5: Relationship between INVENTORY_PART view, INVENTORY_PART_CFT table, and INVENTORY_PART_CFV view.

means that a record in the INVENTORY_PART may be associated to one record in INVENTORY_PART_CFT table, but it is not necessarily. The circle symbol at INVENTORY_PART_CFT means that every record in the table is associated to a one record in INVENTORY_PART view.

INVENTORY_PART view has similar association to INVENTORY_PART_CFV as to INVENTORY_PART_CFT. INVENTORY_PART_CFV and INVENTORY_PART_CFT have a mandatory 1:1 relationship, as for every record in INVENTORY_PART_CFV view there will be a record in INVENTORY_PART_CFT table. This is because the INVENTORY_PART_CFV view will only contain data if the record has custom attributes. If it has no custom attributes, the record is only available in INVENTORY_PART view. Technically, INVENTORY_PART_CFV is a view that combines the INVENTORY_PART_CFT table and INVENTORY_PART view by their unique identifiers. Records in INVENTORY_PART_CFV view are a combination of all

attributes from these INVENTORY_PART view and INVENTORY_PART_CFT table.

For the implementation, five dummy custom fields were created to the inventory part entity. The implementation is split into two parts: first is to load the first 100 rows from INVENTORY_PART_CFV view without doing any additional changes to the view or its data. Second part is to add an sixth custom attribute in to the view and see if it has any consequences to how the REST API or the JSON works. After that the new custom attribute is deleted to see if it has any consequences to how the REST API or the JSON works. By assumption, a well-created REST API and dynamic JSON can cope with modifications without having to be manually maintained.

First part

The first test simply ensures that the API supports custom objects and tests how custom objects are displayed in the response JSON. To test this, an API that handles INVENTORY_PART_CFV data was found from API Explorer, which was presented in Figure 3.2. Then in an API management platform a request according the Odata syntax was sent to the API, where the first 100 rows are retrieved from the database. The amount of rows retrieved is arbitrary, and not related to the test or the functionality of the API in this case.

The JSON response was evaluated to address the problems. Two conclusions could be drawn. First, the API supports custom attributes, as desired. This is a good thing, because it is a prerequisite for the second test of this implementation. Second, custom attributes are in JSON in the form of key-value pairs, just like regular columns. This is especially important for further processing of JSON. Another way to represent custom object data would be, for example, to return the key "custom objects" in JSON with the value of a list of custom attributes as key-value pairs.

Further processing of such data is by default more difficult and would require more configuration.

Second part

After verifying that the REST API supports custom attributes according to our needs, it is possible to test if adding a new custom attribute to INVENTORY_PART_CFT table affects the REST API. To create a new custom attribute, one needs to use IFS Entity Configurations page, search for the entity to be configured, i.e. inventory part in this case, and from the GUI add a new custom attribute. After configuring the new custom attribute one needs to synchronize the new attribute with the database, which means that the inventory part entity will be re-deployed to the database.

After the deployment has been successfully completed, same steps as in this implementations first part were done. Same request was sent from the API management platform to the REST API and the resulting JSON was evaluated. Everything went as planned and the response JSON had the new custom attribute. In the database the new custom attribute was added to the INVENTORY_PART_CFT table, as expected.

IFS entity configurations page is used again to delete the custom attribute. The removal is done by pressing a delete button in the GUI, after which the custom attribute is removed from the custom attributes list. After deletion, the IFS indicates that a cleanup is still necessary to remove the custom attribute from the database. At this point, the custom attribute is no longer visible to the end-user in practice, because the response JSON from REST API does not return the deleted custom attribute and it is not visible in the GUI anymore.

When querying in the database for the INVENTORY_PART_CFV view the custom attribute has been deleted, but for the INVENTORY_PART_CFT table

it is still retrieved. In theory, this could cause problems in some cases if cleanup is not done. However, end-users usually do not use the database connection, or if they do, they are more likely to use INVENTORY_PART_CFV view instead of INVENTORY_PART_CFT. Similarly, the same view would be used for report generation and data warehousing, for example, so that a hanging attribute would not cause problems. After the cleanup has been triggered and finished from the IFS it was also removed from the INVENTORY_PART_CFT table. Thus, completely deleted from the IFS and its database.

3.2.3 Transferring data from multiple sources with one load

As third and last implementation, we will implement a workflow that transfers data from multiple sources with one load. The idea of this implementation is to resemble how a real data warehouse would work in IFS environment using only REST APIs. For this, we have created a Azure data factory workflow for data loads that has access to both source system and target database. The goal of the implementation is to verify that data can be transferred from multiple REST API sources to multiple database tables without issues.

The results of this implementation are crucial for answering RQ2 and RQ3, as it represents the closest version of a minimum viable product developed in this Thesis. Firstly, the implementation represents how data loads could be done efficiently to SaaS source systems, which contributes one possible solution to RQ2. Secondly, any limitations with REST interfaces in the implementation provides insights into their impact on data warehouse maintenance regarding RQ3. The evaluation of this implementation will determine how well these technical decisions support the overall idea of creating data warehouse solutions solely based on REST APIs.

The main difference between this implementation and the first is that this implementation is designed to handle multiple data sources to data warehouse while

the first implementation explores how REST API data loads work from a single large source. The main challenge in this implementation is how to retrieve data from multiple REST APIs and transfer it to multiple target tables in a single data warehouse efficiently. The data used is financial data related to accounting in IFS ERP, retrieved from three different REST APIs. Contents of the data are irrelevant to the implementation and same results could be achieved with any IFS provided REST APIs.

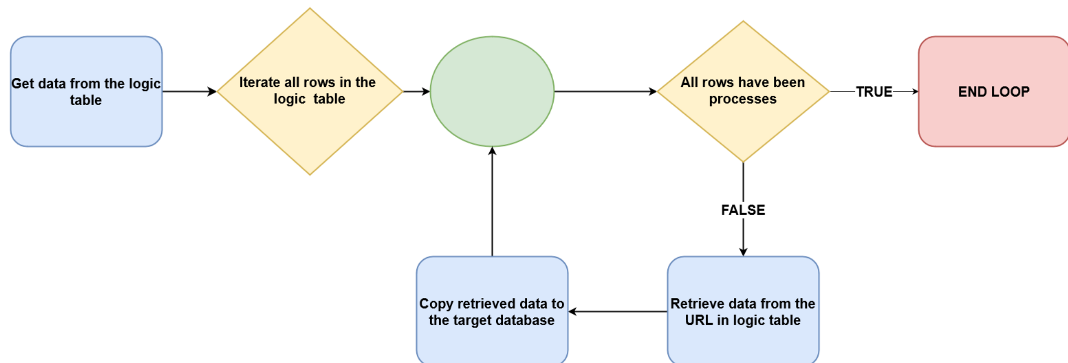

Figure 3.6: Workflow of the data warehousing implementation

The setup for this implementation is similar to the first implementation. The target tables are deployed in advance using the same tool, the data is retrieved using the REST APIs, processed in JSON format, and the data is mapped to the target table using the same script stored in the target database. The main changes compared to the first implementation are creation of a logic table to the target database and the addition of a loop to the Azure data factory pipeline. A more detailed description of the workflow can be seen in the Figure 3.6.

The logic table consists of REST API URL used to acquire the data, target table schema, and target tables name. An example of logic table with similar rows can be seen in Table 3.3. The logic table is used to make looping multiple data sources to target tables possible. This allows the implementation to be scaled up, as

REST_API_URL	TARGET_TABLE_SCHEMA	TARGET_TABLE_NAME
Rest API URL for account data	ifsinfo	account
Rest API URL for account group data	ifsapp	account_group
Rest API URL for account type data	ext_ifs	account_type

Table 3.3: Example of logic table used for transferring data from multiple REST API data sources

multiple such tables can be created for different scenarios. For example, such tables could be created for account and customer data that needs to be stored. This is important because it allows for variance in warehousing times. For example some implementation would want to only store account data weekly and customer data daily.

The new loop added to the Azure data factory loops over the values in the logic table. For each row, a mapping is first retrieved from the target database using the data in the TARGET_TABLE_NAME column. The data is then retrieved from the url in the row and copied to the target database by using the mapping. This continues until all rows in the logic table have been processed, i.e. in this case until all the rows defined in the account logical table have been copied from the source system to the target system.

The loop goes through all the logic tables whose names are given to it as parameters. It is therefore easy to create a loop that iterates through only a given set of logic tables. Another way to do this would be to create a new logic table where the tables belonging to a particular load would belong to the same ID. For example, all tables that are loaded every 24 hours would have a specific ID in the logic table. This is one possible approach for environments where there are many different time intervals between data loads and tables to be loaded.

Alternative way to achieve this would have been by using one single logic table with IDs. In a logic table made in this way the IDs would represent a specific entity's data, for example ID 1 for all account data, ID 2 for all customer data etc. The

benefits of this method would be, for example, the ease of transferring all the data, as the user would only have to iterate through the table, ignoring the IDs. On the other hand, the single table would grow in size and it would possibly be harder to maintain. In this implementation, entity-specific logic table was made precisely for maintainability.

4 Results

This chapter presents the results of the evaluation of the various REST API data warehouse test implementations developed for this study. The purpose of this analysis is to assess how REST APIs change data warehousing implementations, how data loads were implemented through REST APIs, and how REST API data loads impact the maintenance of the data warehouse. Together the analysis shows the strengths and weaknesses of a data warehouse using REST interfaces compared to conventional SQL-to-SQL data loading methods.

The results of this study are examined implementation by implementation in this section and summarised in more detail in the final chapter. The results of the different implementations are examined in the order in which they are presented in Chapter 3. In general, implementations are compared to similar SQL-SQL data warehousing implementations. However, the comparison is not always fair because, as stated earlier, the end user may not have access to the database, in which case data warehousing via the REST API could be the only option.

4.1 Implementation 3.2.1

The implementation 3.2.1 consists of REST API loads to a single database table. The goal of the implementation was to test how REST API interfaces can handle different amounts of data. The test was performed with data loads of 100,000, 250,000, 500,000, 1,000,000, and 2,000,000 rows from a single database table, focusing on the

differences between different row counts. In addition, the size of JSON data loads was compared to SQL data writes. From the implementation, it became clear that time and size increase linearly with the number of rows loaded.

Implementation 3.2.1 proves that even large data loads can be done with SaaS source systems. One clear advantage of this is that data loads can be done without SQL knowledge, although one needs to know the API syntax which is arguably easier to handle. When syntax is easier to learn and manage, it is also easier to setup and maintain API loads. On the other hand, loads made via the REST API require a JSON to SQL mapping, as the data received from the data load is in SQL format. This is unlike SQL-SQL loads where the data received is in SQL format already, but which generally require more programming skills compared to REST API loads.

The main result from the implementation is that data loads from smaller databases are a perfectly valid way to implement data warehousing. However, as the amount of data grows, more efficient data types than JSON are needed. One of the ways to do this is to use compressed JSONs instead of regular JSONs that lowers the amount of data transferred by a lot. This is not always possible, as in this implementation, because the API provider did not support such a feature. If the API provider does not support such functionality, the end user has no choice but to settle for the standard JSON loads. Standard JSON loads will probably become a bottleneck in larger REST API data loads, unless the time limit is long. 5

Based on these results, an indicative table of which data loading method could be used at any given time was created. Table 4.1. shows the recommendations for the different data loading types in relation to the size of the database in millions of rows. The table does not take into account the fact that the REST API may not be available or, alternatively, that no access to the database is available to restrict data downloads. However, the table shows that REST API downloads without compression are not suitable for very large databases. For example, based on Table

3.1 in the implementation, a 10 million row load via REST API would take about 11.5 hours.

	0-1M	1M-5M	5M-10M	10-20M	20M<
REST API loads without compression	x	x			
REST API loads with compression	x	x	x	x	
SQL Loads	x	x	x	x	x

Table 4.1: A table showing what data load type could be used for different sized databases.

One way that large amounts of data retrieved from a single REST API endpoint could be handled is by performing parallel data loads. In simplified form, this could be achieved by partitioning the dataset into smaller subsets for data retrieval, for example by downloading the first 10,000 rows in one load, the next 10,000 rows in another load, and so on. After the data has been loaded, the write operation can be performed, allowing for faster overall integration times. In theory, this approach could enable the transfer of very large datasets, provided that the number of rows loaded per REST API call is appropriately configured. In practice, such large-scale data loads would be more difficult to implement and more prone to errors, which is why this approach is considered outside the scope of this thesis.

4.2 Implementation 3.2.2.

The implementation 3.2.2. focused on more source system specific case, where IFS custom attributes and their integrability was examined. The goal was to find out wether or not IFS REST APIs support REST API loads and if there are any downsides of using REST API loads compared to SQL loads. In addition to REST API functionality the IFS GUI connection to database was explored, as a case of hanging attribute was figured out. This could cause some problems with badly made SQL loads but it should not affect REST API loads.

By testing the REST API with custom attributes, it can be said that the IFS custom attributes are accessible and well-maintained for use with the REST API. Custom attributes are returned in a structured format and are easy to process. These are particularly important when working with data warehousing solutions. The main result is that custom attributes can be treated as normal attributes in the IFS REST API, without any additional effort.

From an integration maintainability perspective, using REST APIs can be considered preferable to direct database access in IFS environments. A single REST API endpoint can be used to retrieve entity data both before and after custom attributes are introduced. This allows integrations to start consuming data for an entity without custom attributes and later include additional attributes without changing the REST API URL. As a result, changing database tables schema caused by custom attributes does not require changes to the integration endpoint, improving long-term maintainability.

Compared to REST APIs direct database access becomes more complex when custom attributes are introduced. Before custom attributes are added, data is typically retrieved from a standard database view, such as inventory_part. Once custom attributes are enabled, these fields are only accessible through a corresponding view with the _cfv suffix, for example inventory_part_cfv, as illustrated in Figure 3.5. As a result all data sources and queries must be updated to use the new view in order to include the custom attribute fields. The _cfv view cannot be used before custom attributes are introduced, as it is only created in the database after custom attributes are added to the entity. This introduces additional maintenance effort as data sources need to be updated for data loads to fetch the added custom attributes.

As explained in the implementation, another, in this case worse, way to implement REST API for custom attributes would be simply putting all the custom attributes in to a list of key-value pairs, returned as one key-value pair in the JSON.

This could be a better solution for some cases, but for an effective data warehouse solution this would mean iterating the list and creating mapping rules for keys in the list. This could be laborious and error-sensitive, thus negatively affecting the maintainability of the data warehouse.

4.3 Implementation 3.2.3.

The implementation 3.2.3. was the closest to minimum viable product of the three implementations. The goal was to simply transfer data from multiple sources to same target database with one data load in the integration tool, using only REST APIs. For this, the integration pipeline became the most complex of the implementations. It had logic not only in the Azure datafactory, but also in the database. Those were necessary when creating the pipeline this way, as the loaded JSONs are mapped to SQL for data writes.

The method created proves that data warehousing is possible using only data loaded from REST API as source data. When only warehousing small amounts of data the difference would not be noticeable between SQL-SQL loads and loads made using REST API. This is important in cases where direct database access cannot be provided and where SQL skills are not sufficient.

Results prove that from a simple one table load as in implementation 3.2.1. to multi-table load representing a real data warehouse load is straightforward. As in this case, only two new components needed to be added to a simple Azure datafactory to scale from a single-table implementation to a multi-table implementation. However, it should be noted that to get the single table integrated to data warehouse the implementation already requires several components, such as the mapping function. All new components increase the need for maintenance and make the job more complex, thus affecting maintainability.

For larger amounts of data, the same problems as in implementation 3.2.1. will arise. As a difference to a single large data load, a user could potentially be able to take advantage of parallel data loads, where multiple tables load data at the same time to the same target, more easily. Parallel data loads would solve the problem in cases where there is a large amount of data, but the data is spread over a large number of different tables, not just one table.

Other similar maintainability challenges still persist in REST API based data warehouse implementations as in traditional SQL based implementations. Such maintainability challenges include adding new entities to the integration, addressing database schema-related issues in data loads, and for example, performance improvements to the pipeline. In the REST-based approach, additional complexity is introduced due to increased integration logic, as the integration requires JSON-to-SQL mapping.

However, the use of REST APIs introduces an additional abstraction layer between the source system and the data warehouse. This abstraction can reduce the direct impact of underlying database schema changes on integrations. While such schema changes are relatively uncommon in mature software systems, this may still be beneficial in environments where direct database access is restricted, with immature software, or in cases where database schema is changed in larger updates to the software.

5 Conclusion

The widespread adoption of cloud computing has been a general trend of recent years. This Master's Thesis presented the main components related to the change in data warehousing brought by cloud computing as a literature review. The literature review showed that there is a lack of literature examining this very problem.

In addition to the literature review the Thesis also presented a one possible way of implementing an integration of SaaS source system to cloud data warehouse. The results of the implementation show that, although integration from SaaS source system to cloud data warehouse using REST APIs has some unsolved problems, it is possible. Assuming that the amounts of data to be transferred are very small, it is also a potential alternative method for regular SQL to SQL loads.

The first research question examined how SaaS source systems change data warehouse implementations. With the growing popularity of SaaS systems, data warehousing is increasingly performed through REST APIs rather than direct database access. As the implementations of this Thesis have shown, this change can reduce performance due to factors such as increased latency and general changes to the ETL pipeline, while also reducing users' overall control over changes in the source system. As a result, data warehouse implementations become more dependent on design decisions made by SaaS providers, particularly regarding REST API interfaces.

These limits are more visible in the export phase of ETL pipelines. SaaS source systems significantly change how data is extracted, as exports are often provided in formats such as JSON rather than SQL, and load times may be considerably slower than in traditional SQL-based integrations. These changes must be taken into account during the transformation and loading phases of the ETL pipeline, as JSON, for example, must be converted into a loadable format such as SQL. Together, these factors affect the overall data warehouse implementation, as more variables must be managed compared to traditional on-premise or database-to-database approaches.

Regarding the second research question, the results show that data loads from SaaS source systems can be implemented by using REST API–based ETL pipelines. In the case of this Thesis REST based API loads were the only available option to access the data of IFS SaaS source system. The REST API loads follow the ISO/IEC approved Odata technical protocol, while Odata Kubernetes pod is responsible of the REST API call receiving on the middle tier server. As a result, data loading from SaaS source systems is primarily limited by the characteristics and constraints of REST API interfaces, requiring ETL pipelines to be designed around API-specific limitations.

The third and last research questions examined the impact of REST interfaces on the mainetenance of the data warehouse. As said, while using REST APIs for data loads users are more prone to the changes initiated by the SaaS vendor. Although not very common, REST API addresses for data retrieval may change, requiring the REST API address used for data retrieval to be updated. Also as mentioned, the data warehouses ETL logic becomes more complex after adding additional components to handle REST API data loads. An example of such component is the transform script used to map JSON data to SQL for data writes. Altogether, these are notable new components related to data warehouse maintenance resulting from the change in data loading.

5.1 Limitations

The implementation focuses to a specific research area, which is strongly influenced by the chosen SaaS system, i.e. IFS and the integration platform, i.e. Microsoft's Azure Data Factory. Using different SaaS source systems or integration platforms could produce different results, as the different options were not explored during the work. On the other hand, it was the use of the IFS as a source system that was key to the client of the study. Despite this, the work could have addressed which integration platform would have provided the best results for the chosen SaaS source system.

Also, the created integration product was only tested with three different use cases. The implementations tested the product's functionality in edge cases where failure would have prevented the implementation of REST API integrations altogether. The data loads done through REST API can only be tested to a certain extent with these limited implementations. More implementations would have provided a broader understanding of the current situation, its problems and new ideas to explore for future research.

The study is also strictly limited to one demo environment, and the minimum viable product has not been tested in test or production environments. In theory, data transfer times could be even longer in production environments, at least in cases where the production environment is in active use at the time of data transfer. As a result, the suitability of the proposed approach for large enterprise environments is not fully verified.

Regarding RQ3 one must also note that the assesment of maintenance is not observed over time on the demo environment where implementations were tested. Instead, the impact of maintenance has been inferred based on observations made during implementations. By continuing the test over several update cycles, for example, maintenance-related observations could be either confirmed or rejected.

5.2 Future work

As already mentioned in the limitations section of the study, one of the weaknesses of this study is its focus on a single SaaS system and integration platform. It is easy to extend the research either by adding different SaaS systems to the comparison, thus clarifying the testing of different APIs in data warehousing. Or by adding different integration platforms to the comparison, allowing a comparison of their weaknesses and strengths with the same API. Or by comparing different databases for data warehousing.

In the future one key research topic could be how to use the REST API to retrieve millions of results efficiently. One option is to reduce the payload, as seen in this work by indexing the search to X number of search results per data load. However, currently indexing does not solve the whole issue, although it makes data loads more efficient on a general level. Another option for payload reduction is payload compression, for example, which was explored but not supported by the REST API used. Another way to solve the problems would be to try some sort of optimization to reduce the payload sizes.

As mentioned in Section 4.1, parallel data loading can be used as a solution for loading large amounts of data not only from a single REST API endpoint but also from multiple REST API endpoints. In the context of IFS, REST API loads could be targeted, for example, at IFS modules. Modules in IFS represent different functional areas within the application, such as production, procurement, finance, and human resources management. By loading data on a module level, the amount of data per load would be smaller, data loads could be started simultaneously, and individual pipelines could progress independently. As a result, the amount of data processed per pipeline would be significantly smaller than in a single large pipeline. Module-specific parallel data loading is illustrated in Figure 5.1.

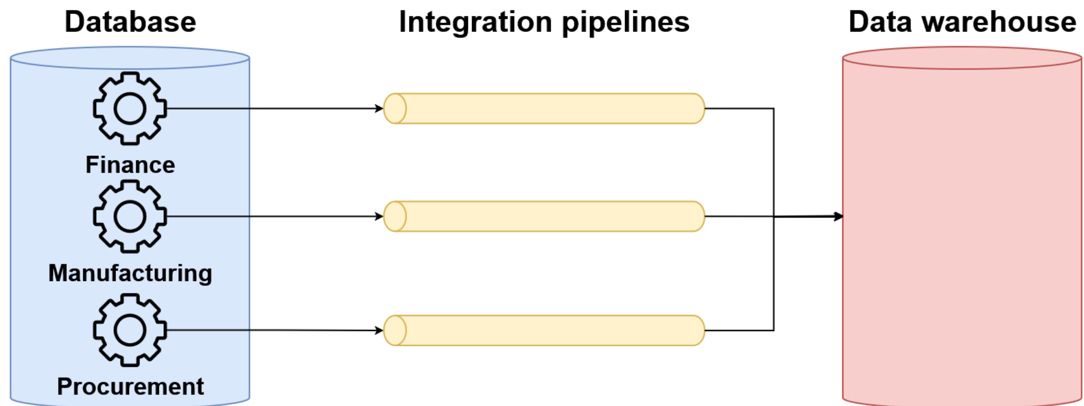

Figure 5.1: Illustration of how data could be loaded in IFS modules to data warehouse to reduce data amounts per integration pipeline

Another approach to handle slow data loads is the use of incremental loads. In the implementations, only full data loads were tested, where all available data was loaded from the REST API endpoints. In practice, this is often not necessary, as a large portion of the data is typically already present in the data warehouse from previous loads. IFS provides a useful database field, objversion, which indicates the last modification time of a row and could be used to support incremental loading. In such a case, periodic loads would only include data that has been created or modified since the previous load.

Incremental loading could also be combined with module-specific parallel data loads. This would further reduce the amount of data processed within a single pipeline, making REST API–based data loading more efficient and a more viable alternative to SQL-based data loads. In this approach, data loading would first be divided by module and then further reduced to incremental changes only. As a possible next step, modules with a large amount of frequently changing data could be further optimized by introducing multiple parallel pipelines within a single module.

References

[1] E. M. Haryono, Fahmi, A. S. Tri W, I. Gunawan, A. Nizar Hidayanto, and U. Rahardja, "Comparison of the e-lt vs etl method in data warehouse implementation: A qualitative study", in *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, 2020, pp. 115–120.

[2] L. Yessad and A. Labiod, "Comparative study of data warehouses modeling approaches: Inmon, kimball and data vault", in *2016 International Conference on System Reliability and Science (ICSRS)*, 2016, pp. 95–99. DOI: 10.1109/ICSRS.2016.7815845.

[3] A. H. Mousa and N. Shiratuddin, "Data warehouse and data virtualization comparative study", in *2015 International Conference on Developments of E-Systems Engineering (DeSE)*, 2015, pp. 369–372. DOI: 10.1109/DeSE.2015.26.

[4] I. Bojičić, Z. Marjanović, N. Turajlić, M. Petrović, M. Vučković, and V. Jovanović, "A comparative analysis of data warehouse data models", in *2016 6th International Conference on Computers Communications and Control (ICCCC)*, 2016, pp. 151–159. DOI: 10.1109/ICCCC.2016.7496754.

[5] T. Kaur and S. Kamboj, "Descriptive analysis of the cloud computing services and deployment models", in *2023 International Conference for Advancement in Technology (ICONAT)*, 2023, pp. 1–6. DOI: 10.1109/ICONAT57137.2023.10080749.

[6] A. A. Al-Ghofaili and M. A. Al-Mashari, ”Erp system adoption traditional erp systems vs. cloud-based erp systems”, in *Fourth edition of the International Conference on the Innovative Computing Technology (INTECH 2014)*, 2014, pp. 135–139. DOI: 10.1109/INTECH.2014.6927770.

[7] Gartner, *Gartner forecasts worldwide public cloud end-user spending to reach $679 billion in 2024*, Last accessed 15 January 2024. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/11-13-2023-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-679-billion-in-20240.

[8] P. Appandairajan, N. Zafar Ali Khan, and M. Madiajagan, ”Erp on cloud: Implementation strategies and challenges”, in *2012 International Conference on Cloud Computing Technologies, Applications and Management (ICCCTAM)*, 2012, pp. 56–59. DOI: 10.1109/ICCCTAM.2012.6488071.

[9] D. Kapil, P. Tyagi, S. Kumar, and V. P. Tamta, ”Cloud computing: Overview and research issues”, in *2017 International Conference on Green Informatics (ICGI)*, 2017, pp. 71–76. DOI: 10.1109/ICGI.2017.18.

[10] J. Surbiryala and C. Rong, ”Cloud computing: History and overview”, in *2019 IEEE Cloud Summit*, 2019, pp. 1–7. DOI: 10.1109/CloudSummit47114.2019.00007.

[11] Nitu, ”Configurability in saas (software as a service) applications”, in *Proceedings of the 2nd India Software Engineering Conference*, ser. ISEC ’09, Pune, India: Association for Computing Machinery, 2009, pp. 19–26, ISBN: 9781605584263. DOI: 10.1145/1506216.1506221. [Online]. Available: https://doi.org/10.1145/1506216.1506221.

[12] B. B. Rad, T. Diaby, and M. E. Rana, ”Cloud computing adoption: A short review of issues and challenges”, in *Proceedings of the 1st International Con-*

ference on E-Commerce, E-Business and E-Government, ser. ICEEG '17, Turku, Finland: Association for Computing Machinery, 2017, pp. 51–55, ISBN: 9781450352482. DOI: 10.1145/3108421.3108426. [Online]. Available: https://doi.org/10.1145/3108421.3108426.

[13] G. C. A. Peng and C. Gala, "Cloud erp: A new dilemma to modern organisations?", *JOURNAL OF COMPUTER INFORMATION SYSTEMS*, vol. 54, no. 4, pp. 22–30, 2014, ISSN: 0887-4417. DOI: 10.1080/08874417.2014.11645719.

[14] V. Goar, P. S. Sarangdevot, G. Tanwar, and D. A. Sharma, "Improve performance of extract, transform and load (etl) in data warehouse", *International Journal on Computer Science and Engineering*, vol. 2, May 2010.

[15] K. T. Shishmano, V. D. Popov, and P. E. Popova, "Api strategy for enterprise digital ecosystem", in *2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC ST)*, 2021, pp. 129–134. DOI: 10.1109/PICST54195.2021.9772206.

[16] G. Bondel, A. Landgraf, and F. Matthes, "Api management patterns for public, partner, and group web api initiatives with a focus on collaboration", ser. EuroPLoP '21, , Graz, Austria, Association for Computing Machinery, 2022, ISBN: 9781450389976. DOI: 10.1145/3489449.3490012. [Online]. Available: https://doi.org/10.1145/3489449.3490012.

[17] M. Coblenz, W. Guo, K. Voozhian, and J. S. Foster, "A qualitative study of rest api design and specification practices", in *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2023, pp. 148–156. DOI: 10.1109/VL-HCC57772.2023.00025.

[18] C. Rodríguez et al., "Rest apis: A large-scale analysis of compliance with principles and best practices", in *Web Engineering*, A. Bozzon, P. Cudre-Maroux,

and C. Pautasso, Eds., Cham: Springer International Publishing, 2016, pp. 21–39, ISBN: 978-3-319-38791-8.

[19] IFS, *Ifs about page*, Last accessed 14 December 2023. [Online]. Available: https://www.ifs.com/about.

[20] IFS, *Ifs 2024 finance*, Last accessed 11 March 2025. [Online]. Available: https://www.ifs.com/news/earnings/ifs-announces-outstanding-2024-financial-results.

[21] IFS, *Ifs cloud overview*, Last accessed 11 March 2025. [Online]. Available: https://www.ifs.com/ifs-cloud/ifs-cloud-overview.

[22] IFS, *Ifs restful apis documentation*, Last accessed 14 December 2023. [Online]. Available: https://docs.ifs.com/techdocs/21r2/010_overview/240_integration/400_rest_apis/.

[23] IFS, *Ifs odata documentation*, Last accessed 15 December 2023. [Online]. Available: https://docs.ifs.com/techdocs/24r2/040_tailoring/300_extensibility/040_ifs_odata/.

[24] IFS, *Ifs architecture documentation*, Last accessed 13 December 2023. [Online]. Available: https://docs.ifs.com/techdocs/24r2/010_overview/040_physical_architecture/.

[25] IFS, *Ifs middletier architecture overview*, Last accessed 12 April 2025. [Online]. Available: https://docs.ifs.com/techdocs/24r2/070_remote_deploy/010_installing_fresh_system/010_planning_installation/007_middle_tier/010_overview/.

[26] IFS, *Ifs aurena documentation*, Last accessed 29 March 2025. [Online]. Available: https://docs.ifs.com/techdocs/24r1/060_development/022_user_interface/030_aurena_dev/010_aurena_overview/.

[27] IFS, *Api explorer documentation*, Last accessed 12 December 2023. [Online]. Available: https://docs.ifs.com/techdocs/22r2/040_tailoring/300_extensibility/020_api_explorer/.

[28] IFS, *Custom attributes documentation*, Last accessed 19 April 2025. [Online]. Available: https://docs.ifs.com/techdocs/24r2/040_tailoring/225_configuration/400_entity_configurations/100_create_custom_attribute/.