Åbo Akademi

Cloud Migration to Azure Logic Apps: A Case Study Using the Cloudstep Decision Process

Marcus Skrifvars 38514
Master's thesis in Computer Engineering
Supervisor: Marina Waldén
Åbo Akademi University
Faculty of Science and Engineering
2022

Abstract

Owing to the growing popularity of cloud computing, many companies are moving their applications to the cloud. Aveso OY is an IT company looking to migrate its on-premises integration software to the cloud. Microsoft's cloud-based integration service Azure Logic Apps is chosen as the target platform. While migration tutorials are abundant, models for analyzing the suitability of cloud services are scarce. For the thesis, the Cloudstep migration decision process is used to investigate the compatibility between Azure and Aveso's application. Cloudstep is relatively old and intended for general use. Therefore, its usefulness for modern cloud services is investigated. After performing the decision process and creating a small-scale pilot project in Azure Logic Apps, both the cloud platform and Cloudstep are evaluated. On Azure, Function Apps is the superior option to Logic Apps for Aveso's integration application. Cloudstep has several benefits, namely the clarity provided by profiles. Its weaknesses involve redundant steps and the difficulty of creating an accurate cloud provider profile during the first iterations. By encouraging testing before starting the analysis, Cloudstep better fits Azure.

Keywords: Cloud computing, cloud migration, data integration, Cloudstep, API, Azure

Preface

I want to thank my supervisor Maria Waldén for her valuable guidance during the lengthy writing process. Furthermore, I want to express my gratitude to Sami Heino at Aveso for providing me with an opportunity and resources to work on this thesis.

Marcus Skrifvars
Turku, 23 January 2022

Table of contents

Abbreviations

API	Application Programming Interface
AvIF	Aveso Integration Interface
AWS	Amazon Web Services
ELT	Extract, Load, Transform
ETL	Extract, Transform, Load
ERP	Enterprise Resource Planning
HTTP	Hypertext Transfer Protocol
IaaS	Integration-as-a-Service
JSON	JavaScript Object Notation
PaaS	Platform-as-a-Service
REST	Representational State Transfer, often synonymous with RESTful APIs
SaaS	Software-as-a-Service
SDK	Software Development Kit
SLA	Service Level Agreement
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
XaaS	Anything-as-a-Platform
XML	Extensible Markup Language

1. Introduction

There is no doubt that cloud computing has gained tremendous popularity during the last decade. The shift to cloud services has benefited small businesses and large companies alike due to lower up-front costs and a higher level of scalability. By transferring maintenance over to a cloud provider, the threshold for accessing high-end hardware from anywhere in the world is now lower than ever. Working from home has become a regular occurrence in the past few years, and cloud computing's benefits extend to remote working. Flexibility and greater security compared to saving data locally on employees' devices are only some of the advantages that have become visible due to widespread remote work. Given the surge in popularity of cloud services, it is easy to look at cloud computing through rose-tinted glasses. Moving an existing application to the cloud is not an easy task, especially considering the number of distinct cloud providers that are available, all with their strengths and weaknesses. As seen in this thesis, it can be unclear what services to use on the same platform. A topic that is ignored in many cases is investigating whether an application benefits from cloud hosting, i.e., if cloud migration is necessary at all.

The IT company Aveso currently manages several programs that combine data from separate sources for various corporations. One of these so-called integration services has been selected as a prime candidate for a cloud migration experiment. There are numerous guidelines for moving applications to the cloud, both official documentation by provider companies and paid consultation services by third-party corporations. A common factor for most guidelines is that the platform's suitability is not questioned, and it is up to the customer to determine it in advance. In this thesis, a cloud migration decision model intended for general contexts will be utilized to analyze the appropriateness of the selected cloud platform.

In addition to investigating how well Aveso's integration application works on Microsoft's cloud service Azure, the thesis' main research question will concern how valuable a general decision process will be in our specific integration case. Analyzing existing migration suitability processes would allow pointing out improvements that can be used for similar scenarios. As no public decision process exists for Azure, the relevant aspects of the results could be applied for potential models in the future.

The thesis will begin in Chapters 2 and 3 with an overview of cloud computing, followed by descriptions of the main aspects of migrating an application to the cloud. In Chapter 4, data integrations will be discussed, focusing on integration structure and types. Chapter 5 will describe the case in closer detail, while the migration process will be detailed in Chapter 6. In Chapter 7, the results of both the decision model and the cloud migration itself will be analyzed. Finally, concluding remarks of the thesis and suggestions for future research and improvements are discussed in Chapter 8.

2. Cloud Computing

Until recently, the standard way for companies to run applications was to install them on their hardware. This way of running services locally, *on-premises,* requires the person or organization to either develop the software themselves or purchase it from a company. Deploying the products to an existing computing infrastructure is usually a costly project, requiring thorough planning, technical know-how, and possibly additional investments to hardware capacity. The high expenses of scaling up the business were the main barrier holding smaller companies from expanding. During the early 2000s, an alternate solution, *cloud computing*, started to develop. Introduced to the masses by Amazon in 2002, Amazon Web Services (AWS) was the first large-scale public cloud service where the provider company supplies the user with computing power and storage. Therefore, users can run applications remotely on the servers without necessitating any active control. Cloud computing has since gained traction, with industry giants such as Microsoft and Google also providing cloud platforms. Spending on cloud services has increased yearly, with forecasts predicting a growth of 23% to $332 billion in 2021 [1]. This chapter will describe the main principles of cloud computing.

2.1 Definition

The National Institute of Standards and Technology (NIST) has the following in-depth definition: “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [2] . Other definitions, including IBM’s [3], furthermore emphasize the fact that the computing resources are accessed via the internet. The term “cloud” stems from how external networks were illustrated, in that services are run somewhere else than the user in a large cluster of objects. Cloud computing merges data from several resources using hardware virtualization (see 2.4.2). The virtualization of the computing hardware allows for dividing the resources amongst several users. Cloud services are not synonymous with cloud computing, as they refer to the business products delivered in real time over the internet using cloud computing. NIST further defines cloud computing using what many call the 5-4-3

principles of cloud computing: Characteristics, deployment models, and service models.

2.1.1 Cloud characteristics

NIST considers the following five characteristics to be essential for any cloud computing model:

1. On-demand self-service: Computing abilities can be provided automatically without control actions by humans.
2. Broad network access: Resources are available via the internet or local area network, with low latency, supporting standard devices such as computers, phones, and tablets.
3. Resource pooling: Multiple consumers can use the same physical infrastructure simultaneously isolated from each other, and resources are dynamically allocated based on demand.
4. Rapid elasticity: The cloud service should scale elastically, providing no limit on capabilities for the user.
5. Measured service: The usage of resources is measured, allowing users only to be billed accordingly. Payment models can range from "pay-per-use" to subscriptions with a set price each month.

In *Essentials of Cloud Computing*, K. Chandrasekaran adds 14 requirements to the list, specifically for cloud services [4]. Many of the requirements touch on the same subjects as NIST's characteristics. However, some notable additions are interoperability and security: It is expected that the cloud environment has established specifications. Additionally, cloud providers must strictly control access to various resources.

2.1.2 Deployment Models

Cloud services can be deployed in a variety of ways, depending on the location and structure of the organization. If the cloud infrastructure is exclusively for internal use, it is *private*. A private cloud gives the company the benefits of cloud computing (see 3.2), except that the IT department is still accountable for managing the cloud [5]. This approach has advantages and disadvantages, as the company does not depend on a third party (the cloud vendor) in case of outages while offering deeper configuration options. The drawback is that running a private cloud requires technological expertise not necessarily found in every organization. If access rights are expanded slightly to allow shared use by several organizations with the same

interests, the term *community cloud* is used. Opposite on the spectrum is *public clouds*, accessible to the general public. Finally, combining two or more of the previously mentioned deployment models creates a *hybrid cloud*. Due to the high popularity of public clouds, when people refer to cloud computing, they often pertain specifically to public clouds. Contrary to private clouds, the cloud service provider is responsible for managing and maintaining the system. As the user base in public clouds is substantial, security concerns are even more paramount compared to other deployment models.

2.1.3 Service Models

According to NIST, there are three standard service models for cloud computing, the first of which is *Software as a Service* (SaaS). Sometimes the SaaS service model is referred to as cloud applications. This service model offers the highest level of abstraction, supplying the consumer with applications running on the cloud provider's servers. The customer does not control or see the underlying infrastructure unless the cloud provider has expressly provided configuration settings for the application. With *Platform as a Service* (*PaaS*), consumers are provided with a development environment, making it possible to create applications. The PaaS vendors are responsible for managing the underlying hardware.

Moreover, PaaS includes several sub-models, for instance, *Data Platform as a Service* (*dPaaS*) and *Integration Platform as a Platform* (*iPaaS*) for data management and integration flows. If a customer demands even more control of the process, they can choose *Infrastructure as a Service* (*IaaS*). With IaaS, consumers are in charge of the operating system, storage, applications, and particular network elements, namely firewalls and virtual local area networks. Using abstraction, low-level settings such as data partitioning are controllable using APIs (see 2.5). Researchers use the term *Anything as a Service* (*XaaS*) as an all-encompassing term to express the increased rate of selling technology as services. Businesses are not confined to the three models proposed by NIST, and today, specific utilities, including databases and security, can be sold as cloud-based services [6].

2.2 Benefits and Disadvantages

The continual growth of the cloud computing market can be attributed to the high number of benefits involved in the process. Many advantages have become even

more evident during the COVID-19 pandemic [7]. Resources on the cloud are by design accessible at any time from anywhere in the world. In addition to the necessary support for remote working, teamwork and overall collaboration are enhanced by shared access to applications and documents. Cloud computing offers a nimbler way to regulate system productivity compared to a traditional on-premises solution. Users can quickly scale the used resources to match the demand to reduce redundant overhead and, therefore, operational costs as well. Capital costs also decrease significantly, as significant investments into hardware and licensing fees are no longer necessary. As the user installs no physical hardware, the time to deploy is likewise shortened, as is the time and work involved with updating cloud applications. There is less need for in-depth knowledge in areas like server installations, minimizing the requirement for specially trained personnel. Cloud services can furthermore be deployed to users worldwide simultaneously.

A considerable drawback involving cloud computing is the requirement of a constant, high-speed network connection for it to function. Even if the user's connection is stable, high traffic on a cloud server can still lead to high response times. Even if the internet connection is not a problem, companies are still faced with the issue of resource ownership and control. Ultimately, users are at the mercy of the cloud vendors, with possible restrictions on the availability and more complex configurations. The two most significant concerns for users moving into clouds are security and data privacy [8]. For companies handling sensitive data, storing information on the cloud can make it more susceptible to cyber threats such as data breaches, data loss, and account hijacking.

It is worth bearing in mind that data safety provided by the cloud can, in some cases, benefit a company, depending on its size and access to security infrastructure. Smaller companies might prefer cloud providers' data safety and backup options with vastly developed security controls compared to their own. Ultimately, it is up to the end-user to decide if they trust their on-premises solution or the cloud. The level of accessibility of cloud services can also be either a positive or a negative based on client requirements. All cloud providers offer *service level agreements* (SLAs) for their services. SLAs are contracts between customers and service providers where aspects such as availability and quality are agreed upon. SLAs can have different tiers based on pricing, with compensations if the agreement is broken. The guaranteed availability of 99.9% is expected for cloud services and is sufficient for most companies, sometimes even better than what corresponding on-premises options would achieve. Nevertheless, the resulting allowed downtime of

44 minutes per month can be a dealbreaker if uninterrupted access to the service is critical.

2.3 Microsoft Azure

Microsoft's cloud computing service Azure [9] was launched in 2010 as a PaaS product, competing against rivaling cloud services Amazon EC (Elastic Cloud) 2 and Google App Engine. The Azure platform initially focused on four pillars: computing services, blob storage, database services, and the Azure Service Bus, a messaging service originating from BizTalk. Azure has since expanded, with the platform now offering over 200 cloud services and products, including solutions for Big Data, IoT and data integration. Relevant tools and applications on Azure will be presented in Chapter 4. Actual subscriber amounts cannot be found on the web, although in 2018, it was reported that Azure gathered 120,000 new subscriptions per month while managing 95% of all Fortune 500 companies on its cloud services. [10]

3. Cloud Migration

Because of the increased appeal of cloud computing, many companies have started to *migrate* their technically outdated applications to the cloud. These are often referred to as *legacy* applications, systems that work as initially intended but do not allow for implementations of newer technology, therefore hindering growth and interactions with modern systems [11]. Migrating a legacy application to the cloud is not a straightforward procedure, and neither is it a "fix-all" solution that automatically reduces costs and maintenance. Every aspect of the migration process must be thoroughly examined in advance. The risks and benefits of the migration should be assessed to ensure that the chosen cloud platform is a good match for the application in question. It might even be beneficial to take a step back and look at the entire organizational profile and end goals of the application to evaluate if cloud migration is even warranted. This chapter will discuss cloud migration models and strategies, followed by an example model [12].

3.1 Migration Models

Cloud migration models are structured methodologies to help users make informed migration decisions and choose a cloud platform. Migration models have been created in the realm of academics for public use but have also become a service for specialized companies. Cloud providers, namely Microsoft and Amazon, have published their migration models [13] [14] that are to some extent tailored to their services but nevertheless valuable for general cases. Although migration models can vary in business area and scope, they generally contain the following steps in one shape or form: Assessment, planning, execution, validation, and operation. Depending on what is emphasized in the migration model, steps can be missing, combined, or further divided into more parts. The iterative approach is common for all models: After the migration is executed, goals can be set for the next iteration after evaluating the results.

The first phase, assessment, generally involves selecting a cloud provider, reviewing the purpose of the migration, performing technology and business analyses followed by a workload estimate. Migration to the cloud can be motivated by a variety of triggers, including changes in capacity needs and cybersecurity threats. Therefore, it is reasonable to investigate how significant the benefit of using cloud computing is in concern to the trigger. Business analysis can be done using

metrics such as the return of investment (*ROI*). In contrast, an analysis of the technology is helpful for discerning architectural and technological differences between the on-premises application and the cloud environment. Once an assessment has been completed, planning can commence. A suitable *migration strategy* (see 4.2) is selected in this phase, and necessary steps to make it possible are defined. Required changes to network infrastructure, code, and dependencies are among the elements that must be considered. The migration can be executed using the selected strategy when a plan has been established and approved. Validation and optimization are continuous processes that start when the migrated application has been deployed. Performance and functionality are compared to expected results expressed in the assessment and planning phases. Improvements and fixes can then be implemented iteratively. Finally, operating the finished cloud service demands governance in monitoring and securing cloud resources.

3.2 Migration Strategies

A cloud migration strategy is a plan of how an application will be adapted to the cloud. In the e-book *Migrating to AWS: Best Practices and Strategies* by Amazon, six different strategies are presented, referred to as "the 6 R's" [14]:

I. **Rehost.** According to Amazon, most applications are rehosted in large legacy migration scenarios where implementation speed is crucial [14]. This method is often called "lift-and-shift" because the application can be moved to the cloud without any changes to code or architecture. A basic rehost does not automatically bring all the benefits of cloud computing, as legacy applications are not optimized for the cloud, nor are they as scalable as applications native to the cloud. Rehosting can be done via tools provided by Azure and Amazon.

II. **Refractor.** The most expensive and time-consuming strategy is to modify the application to better suit the cloud environment. Re-architecting can require substantial changes to the application code and, therefore, more testing, although it does outweigh the negatives by offering cost savings over the course of time.

III. **Replatform.** Replatform or "lift-tinker-and-shift" can be seen as a middle ground between the strategies above. Replatforming requires managing scope so that the migration does not turn into refactoring and re-architecting. Minor modifications and optimizations are done, but the core architecture remains unchanged.

IV. **Repurchase.** Repurchasing reduces the effort and skill required to carry out a cloud migration. A company can ignore its existing application and instead use a cloud service with corresponding features.

V. **Retire or retain.** Some assets and services can be identified as no longer valuable and can thus be retired and ignored completely. An organization might not be prepared to migrate parts of its services and applications to the cloud, instead preferring to keep them on-premises. In these cases, it is best to reassess the situation later.

These migration strategies are not exclusive. For example, applications can be easier to refactor and re-architect once they have first been rehosted and are run on the cloud [14].

3.3 Cloudstep

Introduced by Patricia V. Beserra et al. in the conference paper "Cloudstep: A Step-by-Step Decision Process to Support Legacy Application Migration to the Cloud" in 2012 [15], *Cloudstep* is a cloud migration model designed to systematically guide application developers and project managers in cloud adoption decisions. Compared to contemporary migration models, Beserra et al. have emphasized the careful assessment of various factors involved with the process that are either risks or benefits. A large part of these factors can be found by identifying the characteristics of the organization, the legacy application, and the selected cloud provider.

As seen in Figure 3.1, the Cloudstep workflow is divided into nine activities. First, the organization profile is defined, and its constraints are evaluated. Administrative and legal characteristics are outlined, provided that they are relevant to the migration. Essential questions such as the main motivations and benefits for the migration will be answered here, in addition to researching if any laws restrict the physical location of the data. The inspection of organizational constraints is done here to pinpoint critical factors hampering cloud adoption at an early stage. IT knowledge of the organization's staff and the location of the data can be listed at this point.

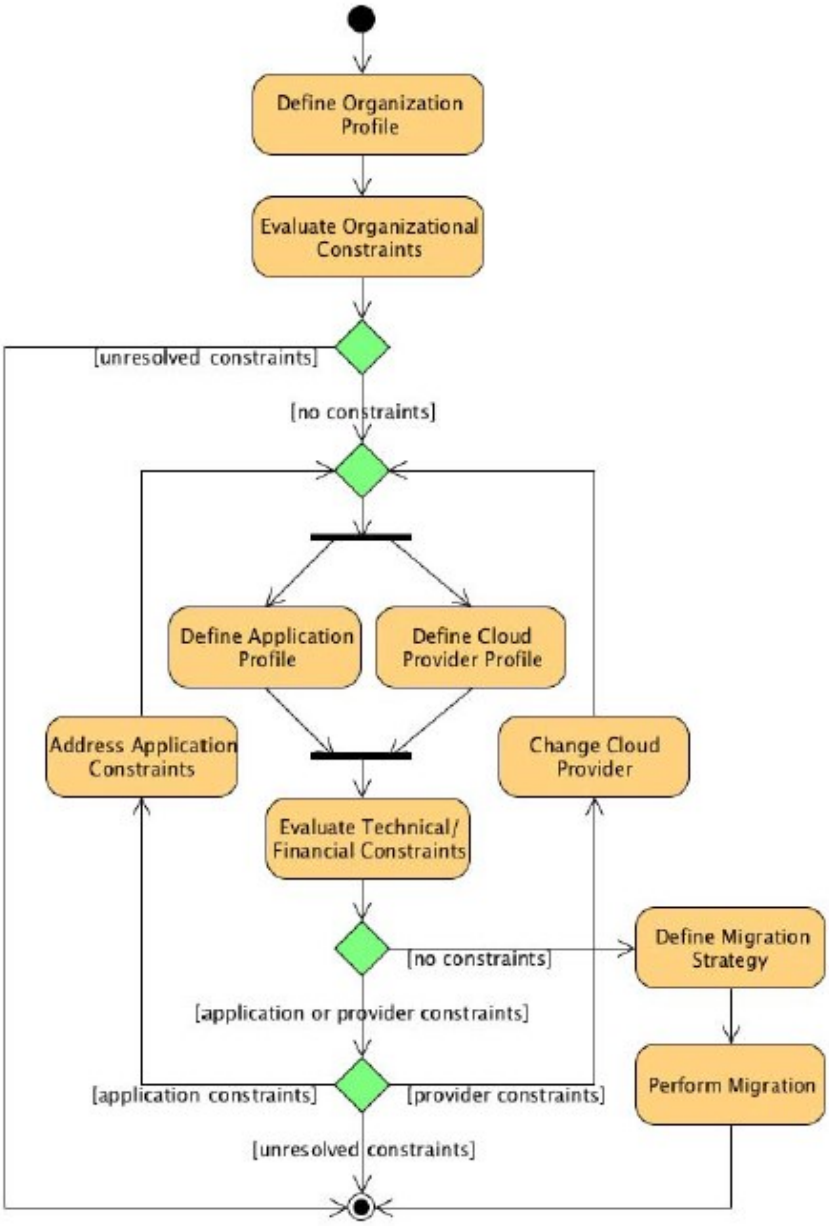

Figure 3.1: Cloudstep workflow [15]

The next step is to define application and cloud provider profiles. The former includes identifying both usage and technical characteristics. The application's main features, user numbers, and patterns are analyzed for usage characteristics. Aspects like technologies necessary for the application and performance requirements are in turn considered when creating technical characteristics. Features such as service models, SLAs, price, and safety are investigated for the cloud provider profile. When both profiles are complete, they can be compared to evaluate technical and financial constraints. Beserra et al. have listed a set of seven constraints that can be used to judge the suitability of the match between a legacy application and a cloud provider:

1. **Financial**: Cost restrictions of operation and performing migration.
2. **Organizational**: Requirements on the skill level of staff, legislation regarding storing of data.
3. **Security**: How well do the cloud provider's security mechanisms conform to the organizations?
4. **Communication**: Restrictions on bandwidth, transfer rate, and latency.
5. **Performance:** Limits on capacity and quality of communication with the cloud provider.
6. **Availability**: Are the offered SLAs sufficient?
7. **Suitability**: Evaluate necessary changes to the application to make it suited for the cloud platform.

If there are no violating constraints, the organization can continue to the next stage, where the migration strategy is selected and executed. If there are unresolved constraints, three possible actions are possible depending on which profile violated the constraints. The violating constraints can be addressed either in the legacy application by modifying it and increasing migration scope or in the provider by selecting another cloud service. The selected profile is created again after addressing the issues in both cases. If the constraints are deemed unsolvable, the process is aborted.

After the migration is performed, the process again checks for unresolved constraints and loops back to the second step. This cycle functions as a feedback loop and allows for iterative development of the cloud migration.

4. Data integration

Data integration solutions become useful when gathering data from several independent sources. Consider the following simplified example:

You have created a website for finding movies to watch. A basic web scraper is used to gather movies to your database, where the movie's title, the director, and the release date are stored. You quickly notice that users might want to access additional information regarding the movies, such as actors, writers, and composers. Adding these rows to your database proves to be a vast undertaking, increasing the size of your data by a considerable amount and making it more challenging to maintain. Furthermore, it has come to your attention that users would like to see the reviews for each film on your website. You would also want to provide links to where the film can be watched. This task turns out to be even more laborious than the previous one; You find that using scores from a single website is insufficient, so a combination of the scores from the five most popular review providers is utilized to calculate an aggregate score. Links to watching the movie are created by parsing all available streaming sites. The database is becoming progressively more demanding to manage. You observe that the new columns must be refreshed frequently. Review scores can change over time, and movies can switch between streaming services or be removed entirely depending on what deals have been struck with the distribution companies. You have realized that this way of gathering and storing data is not feasible, and you are forced to rethink your approach. You are gathering and combining a large amount of data from various sources, with some values such as review scores requiring additional calculations and formatting.

This example illustrates a case that would benefit from using a data integration system. It is worth noting that real-life scenarios are more complex, with hundreds of connections with a vast array of possible data formats.

4.1 Definition and challenges

As businesses grow, they also start producing more data. Massive technology companies such as Facebook are notable for the amount of data their users generate [16], and the growing number of mobile- and IoT devices fuel the surge of data even further. Today the term *big data* is used, meaning exceedingly large data sets that cannot be handled via conventional data processing due to their size [17]. It

has become beneficial for corporations to be intertwined by sharing databases. Combining information from databases requires data to be processed in various ways, as they can often be incompatible with each other in their original formats. Data integration is a set of methods for allowing data providers to share data in its most simple definition.

In "*Principles of Data Integration*", AnHai Doan et al. give a more in-depth description of data integration. According to Doan et al., the goal of a data integration system is "to offer uniform access to a set of autonomous and heterogeneous data sources" [18]. Heterogeneity signifies that the data sources have been developed separately. They can run on different systems, and the provided data do not have to be in the same format. Autonomy describes the fact that the administrative rights to the sources can vary, meaning that the access rights and times can change based on the provider. Doan et al. point out that the primary use case for most integration systems is to access data via queries. However, updating the data in the sources can also be necessary for some instances.

Doan et al. describe the challenges one must face when creating data integration systems. They separate them into system-based and logical obstacles. System-based obstacles express that making systems communicate is not easily achieved. Even though their databases are in a standardized format, variations in implementation can lead to discrepancies when data are finally combined. There can also be disparities in how quickly the data sources can process queries. Logical obstacles encompass variances in database schemata. There can likewise be differences in how the table presents data. For example, one database stores users' first and last names separately while another database stores them as one "full name" field. This inconsistency makes matching the users between the two databases more laborious. Lastly, Doan et al. name social aspects as the third and final obstacle concerning data integration systems. Merely finding the data in question can be a problem, as the provider might only store a portion of the data electronically. Moreover, obtaining access rights to datasets requires cooperation. Some companies might not want to share their data due to legal reasons, or their data can give them a technological advantage compared to competitors.

4.2 Extract Transform Load (ETL)

The use case prevalent for all data integrations is to move data from one location to another. Before the data are transferred to the end system, they often require formatting to be in an appropriate format. The steps mentioned earlier have been

termed *extract, transform and load* (*ETL*). The concept is the base principle behind all data integrations, with deviations in implementation resulting in a wide range of different processes.

4.2.1 Extract

Before extracting information from a source system, it is imperative first to profile the incoming data, that is., analyze and evaluate the format and content. Profiling tools allow for the creation of validity checks for the source data. It is up to the integration designer to handle invalid data. Incorrect extract data could be discarded or corrected manually in the source domain or as part of the transform later in the procedure. Further thought must likewise be put into how the extracted data are accessed. The information is most often situated in another physical location or organization, meaning that several network security layers need to be passed. Servers that store the integration data often have an application security layer. Fortunately, it is common practice for organizations to have separate networks for internal and external activities, each one separated with a firewall. This separation allows for easier access for the server(s) intended for public use without compromising the rest of the company's network. A standard way to call the source system is using an *API* (application programming interface, see 4.5). The requested data are exported or copied to a staging area that acts as an intermediate point between the source and target platforms.

4.2.2 Transform

What happens in the transform stage largely depends on the target system's requirements. Basic transform processes perform tasks in a linear sequence. One of the most basic procedures is transformations by *mapping*. When fields have a consistent format in both systems, matching parts can be translated via predetermined rules. Examples of mapping are altering the file format from JSON to XML and changing the format of all Boolean fields to integers, such as from "True" to 1.

Further data might also have to be gathered using the extracted data in a *lookup* action. For example, the extracted data contain the user IDs, whereas the target system expects to be supplied with usernames and emails. In this case, the required information could be fetched by querying a user database using the ID as an input. An expected value can also be based on several lookups, where the retrieved data must be normalized or aggregated to be in the appropriate format. Calculations and conversions using the extracted data can also be necessary if the target value is derived from several extracted data fields. Other possible actions are

transposing and splitting of rows and columns. Finally, more complex data validation can be performed in the transformation stage in addition to checking for null values and removing duplicate rows.

4.2.3 Load

The final step is transferring the extracted and transformed data to the target system, e.g., a database on a server. Target stores can often be accessed via API codes supplied by the system vendor. The database in the end system has its validity checks that the delivered data must pass. How data are added depends on the specifications of the destination system. New rows can be added to database tables, either instantly or incrementally. Some businesses might demand a complete refresh of their tables once new data are available, essentially erasing all earlier records. An audit trail can be employed to keep track of changes in the database, while the performed operations of the integration software can be saved in a separate archive.

4.2.4 ELT and Alternate Methods

Some modern cloud services have switched to an alternate version of ETL, where the last two stages are executed in a reversed order. This method is called *ELT* (extract, load, transform), and it involves loading the raw extracted data directly to the destination system. The main prerequisite for the ELT procedure is that the target database uses *data lakes*. These storage repositories act as the main staging area for raw data to be transformed into structured data. Using ELT eliminates staging in the extraction phase, which is a time-consuming part of the integration process due to the slow f loading and reading data from the disk where it is staged. The flexible nature of data lakes also allows for out-of-order field processing in the transform stage, further improving the execution time. Data lakes started garnering popularity in the mid-2010s, with companies like Amazon and Microsoft now using the technology in their cloud storage solutions [19]. ELT is overall more suited for *data warehouses* and data analysis and reporting systems. Integration speed is crucial for data warehouses due to the large amount of data they process. This version of the ETL procedure can be regarded as widespread in the industry as more and more companies use cloud solutions that support this feature [20].

4.3 Data Integration System Structure and Design

4.3.1 Data Integration Models

In the book *Business Intelligence Guidebook: From Data Integration to Analytics*, Rick Sherman illustrates all the stages of designing a data integration process [21]. Planning starts with a conceptual model where all data sources used in the integration are approved. In addition to data services and databases, source systems can include external or enterprise applications and unstructured files. The next step involves creating a logical model. Here, the sources are again listed, this time on a data set level, meaning that entity/table and file structure are specified. Knowing the format of both the source database and extracted data allows for one or more "success" paths to be set up for the data. Extracted data that do not meet the necessary standards or lead to errors in the transform phase can likewise be directed to a "reject" path. The physical integration model, all sources, targets, and lookups are listed in the third step. On this component-based level, the optimal data flow of the integration environment is also represented. In Figure 4.1, the logical and physical models demonstrated by Sherman are combined, creating a somewhat simplified model for demonstration purposes. Tables and components are clearly defined, as is the order in which they are handled. According to Sherman's best practice methods, reject cases are also detailed for each integration component.

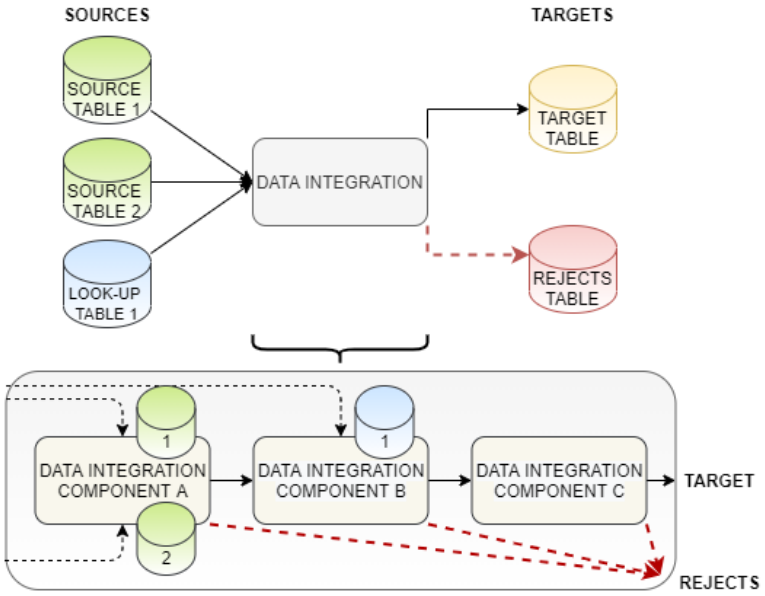

Figure 4.1: Logical and physical model using three sources

4.3.2 Point-to-Point Versus Hub-and-spoke

Conventional *data interfaces*, applications used to move data between systems, are built using a point-to-point method. A system is connected to a single receiver with

point-to-point integration, signifying a 1:1-relationship. In the book *Managing Data in Motion: Data Integration Best Practice Techniques and Technologies* [22], April Reeve labels "hub-and-spoke" design as "the most significant and most important design pattern for architecting real time data integration solutions." Reeve describes how managing traditional point-to-point interactions becomes infeasible when the number of systems grows. If every system wishes to share data via direct communication, the number of interfaces is $(n * (n - 1)) / 2$, where n is the number of systems. The value can be seen as a worst-case scenario, as rarely do all organizational systems communicate with each other.

Nevertheless, the exponential nature of the formula is evident, with ten systems resulting in 45 interfaces, while 100 systems would require up to 4950 interfaces. The term "hub" in hub-and-spoke can be misleading, as it does not refer to centralized data hubs used in business intelligence. Instead, the hub and spoke signifies a system in charge of communications between all systems. Rather than saving data in a repository, hub-spoke architecture requires all data passed among systems to be transformed into a shared format. The typical format is called the canonical language and must be thoroughly planned to support all systems. Using hub-and-spoke design reduces the number of interfaces to the number of systems, n. Only one new interface is necessary for each system, which translates the data to the canonical language, instantly making it compatible with all other systems.

4.4 Integration types

4.4.1 Batch- and real time data integration

There are two predominant approaches to data integrations: *batch* and *real time*. Both have their strengths and weaknesses, making them suitable for distinct situations.

Batch integrations, also called asynchronous integrations, involve grouping data and scheduling the transfers according to a predetermined schedule, such as daily or weekly. The layout and format of the extracted data would be agreed upon in advance between the two parts in order to maintain compatibility. The organization from which the data are sent does not require an instantaneous response for its process to finish. This method of sending data in "batches" is better suited for processing large volumes of data sent consistently. An example of batch integrations is the history page on bank accounts, as it can take numerous days for

credit card transactions to appear on the service. In contrast, real time or synchronous integrations are used in systems that require a response without delays.

Real time data interfaces wait for the transaction to finish in all systems involved before terminating the process. Data sent in synchronous interfaces are generally in the form of compact messages. For instance, when purchasing an item from an online store, the data interface waits for the user's bank account to be updated on the bank service and the order to be confirmed and saved in the online store's database before proceeding. Inconsistencies between the two systems could lead to critical errors. The customer also expects an immediate response to ensure that the purchase was successful. Real time systems have specified constraints in response time and responsiveness. Some organizations want to process data quickly but do not have any precise requirements, with some delays being acceptable. These interfaces are *near-real-time* systems and involve response times in the range of minutes compared to seconds (or below) in conventional real time interfaces. Batch integration is the most cost-efficient type, making it the preferred method to handle non-critical data. The system designers can adjust the integration period according to their preferences, with a refresh interval of one hour theoretically being possible. The variance in implementation can blur the lines between the integration types. Ultimately, the difference lies in the time constraints placed on the data interface and the level of importance they have on the system's overall functionality.

When managing networks with many data interfaces, the terms *loose* and *tight coupling* become relevant. The concepts concern how independent the components in a system are related to each other. It is best to strive for loose coupling while designing interactions between applications and organizations. Loose connections allow a system to fail or be replaced without impacting other systems. According to Reeve in her book *Managing Data in Motion* [22], the API must be clearly defined for one side of the interaction for loose coupling to work in practice. Knowing the methods for all actions, such as requesting functions and storing information for the API, makes replacing one of the systems involved in the interaction achievable. Designing real time data interfaces to allow for applications to be unavailable for a short period essentially makes the interface near real time instead. Batch integrations require tighter coupling as data transfers with large volumes are not necessarily performed via APIs. Changes in either of the systems can require planning from both parties. The coupling of interfaces can, in some cases, be decreased by using standard formats like JSON and XML (see 4.5.5).

4.4.2 Virtualization

A data management approach that is particularly prevalent in business intelligence (BI) is *virtualization*. In essence, data visualization is a subgroup of the data integration concept, employing several integration methods to present all data sources in a consolidated form. As the virtualization layer hides the sources for the consumers, applications can use all provided data without worrying about aspects such as the physical location of the data, database languages, and API definitions. Additionally, using caching mechanisms, virtualization products can reduce the workload contention that a data consumer can cause on data stores when running resource-intensive queries. Data integration and virtualization are not interchangeable terms. While integration is almost always a part of the virtualization process, there are cases where it is not used, for instance, if all data comes from the same data store. The main difference between traditional ETL processes and virtualization is that while the former physically move data from several sources into a centralized data store from where it is accessed, the latter pass on the queries to the source systems without moving any data. [23]

4.5 Application Programming Interfaces

4.5.1 Definition and Types

An *application programming interface*, often referred to in its abbreviated form *API*, is one of the most common ways for software applications to communicate and transfer data between each other. APIs can be perceived as the combination of two components. The first component is the interface, which is visible to external systems. This part contains the specifications to all interactions, often conveyed using internet protocols for web APIs, namely *HTTP* (Hypertext Transfer Protocol) and standardized formats (see 2.5.5). Specifications usually come in the form of documentation. In addition to describing the interface's functionality, the specifications can also contain technical and legal constraints such as rate limits and branding limitations. The second component of an API is the implementation that provides functionality by handling the requested tasks. The strength of APIs lies in their simplicity for users. By virtue of abstraction, developers are only exposed to relevant information, requiring no knowledge about the underlying operations and structures. There are two principal types of APIs: private and public. Both function in identical ways, and only their service goals and legal agreements distinguish them from each other.

Private APIs are used internally by staff and partners in a company. In these instances, the API acts as a front-end interface to access data and application functions in the back end. Private APIs are inherently restricted and thus necessitate contracts for partner companies to access them. Some have further differentiated in-house and partners' APIs by creating additional partner APIs. This type is commonly used in integration scenarios where two parties want to share data.

In contrast, public APIs offer open access to internal resources. Public APIs are accessible to anyone and do not demand any contractual arrangements. Third-party applications can therefore use their features without any licensing fees. APIs can also be separated by use case, ranging from communicating between database management systems and applications (Database APIs) to managing services on a kernel-level (Operating systems APIs). The most common class is web APIs, referring to both web servers and browsers [24] [25] [26].

4.5.2 HTTP Protocol

Web APIs primarily use the HTTP application layer protocol to deliver requests and responses from web applications and servers. An HTTP process starts with a request containing a method, *URI* (Uniform Resource Identifier), and the protocol version. The HTTP methods, also known as *verbs*, are used for specific actions, such as *GET* for retrieving a resource and *POST* for creating a resource. After the request has been processed on the server, the client receives an HTTP response that contains a status code used to diagnose errors, the protocol version, and possibly a body containing requested data. Several API protocols and architectural styles utilizing HTTP have been created to standardize the data exchange between web services. One example is *SOAP* (Simple Object Access Protocol), used to exchange data via XML. Lately, the *REST* (Representational State Transfer) architectural pattern has gained popularity. Web services that follow the architectural constraints of REST are called RESTful APIs and have, in some cases, become synonymous with web APIs in general. [27]

4.5.3 Representational State Transfer

REST was first introduced in the landmark Ph.D. dissertation "Architectural Styles and the Design of Network-based Software Architectures" by Roy Fielding in 2000 [28]. In the dissertation, Fielding uses the term *resource* as an abstraction for any information, be it a file, image, or physical person. A resource can have several formats, called *representations*. For instance, a server could return an item in either JSON or XML format. Resources are identifiable by unique URIs in a format readable for humans. Using the URIs, resources can then be manipulated via HTTP

verbs. These manipulations should always be *atomic*, independent of other processes. Partial resource updates are prohibited in REST, so when updating a resource's field, its entire state must be sent in the request. The website ***restfulapi.net*** [29] lists the six guiding architectural principles of REST; here, the five mandatory rules are recited in a more concise format:

1. User interface concerns must be separated from data storage concerns.
2. Session state is kept on the client. Therefore, all requests to a server must contain all necessary information to interpret it.
3. Response data must be labeled either cacheable or non-cacheable. If the response is cacheable, it can be reused for identical requests.
4. The interface must be uniform. There are four constraints:
    a. Identification of resources is made using URIs.
    b. Resources are manipulated through HTTP methods and URIs.
    c. Messages are self-descriptive and contain all information to process them.
    d. Hypermedia is the engine of the application state. In a genuine RESTful API, all resources carry an explicit or implicit address, allowing REST clients to discover all resources using hypermedia links contained in the response's content.
5. Place constraints on component behavior to allow the architecture to be composed of hierarchical layers.

Aside from the implicit benefits of using a uniform interface, the significant advantages of following REST conventions are increased performance, visibility, and reliability. The separation of client and server allows for tasks such as scaling and editing components to be done independently.

4.5.4 API Specifications

OpenAPI is the most common interface description language (IDL) used in documenting RESTful APIs. The most popular tools used for implementing OpenAPI specifications use the name *Swagger*. In addition to the API's name and version, the OpenAPI specification defines all operations in the interface paired with sample responses. The JSON-formatted OpenAPI specifications are suited for developing new APIs and interacting with existing ones. Swagger developers have added several tools and subprojects, such as *Codegen*, which generates server stubs

and client SDKs based on OpenAPI definitions. Swagger-UI also visualizes the data on an interactive web page, providing documentation and interaction. A screenshot of the API specifications for the IFS REST API used in this thesis can be found in Appendix A. The OpenAPI file is displayed using *SwaggerHub* [30].

4.5.5 Common Formats (XML and JSON)

The two most common formats used when requesting and receiving data using APIs are *XML* (Extensible Markup Language) and *JSON* (JavaScript Object Notation). Both formats are language-independent, allowing them to be used in any programming language. XML files are based on *nodes*, where the node's name describes the attribute, while the data inside the node display the value. Nodes have an opening and closing tag, represented using "<" and ">" symbols. Instead of tags, JSON files use key-value pairs separated by commas to represent data. Tying this subject together with the Chapter introduction, the code snippets in Figure 4.2 are examples of how fetching movie information would look like using the two formats.

The formats have distinct advantages. JSON is deemed by many to be simpler to use due to its smaller file size and higher readability for humans. However, unlike XML, JSON is not self-describing, as the former usually includes links to its schema in the header, allowing for easier validation of the document. In addition to supporting metadata and mixed content, the formatting of XML files enables browsers to render the documents in a discernible pattern. It is important to point out that XML and JSON have different purposes. Compared to JSON, XML is a data format and also a language. XML is used for document markup, while JSON is used for structured data interchange where metadata is not required. JSON is the standard for transferring REST data, making it a hugely popular format. XML is still used for generic API calls such as SOAP. [27] [31] [32].

```xml
<movie_collection>
    <movie>
        <title>2001: A Space Odyssey</title>
        <director>Stanley Kubrick</director>
        <actors>
            <actor>Keir Dullea</actor>
            <actor>Gary Lockwood</actor>
            ...
        </actors>
        <genres>
            <genre>Sci-Fi</genre>
            <genre>Adventure</genre>
        </genres>
        <year>1968</year>
    </movie>
    <movie>
        <title>Goodfellas</title>
        <director>Martin Scorsese</director>
        <actors>
            <actor>Ray Liotta</actor>
            <actor>Robert De Niro</actor>
            <actor>Joe Pesci</actor>
            ...
        </actors>
        <genres>
            <genre>Crime</genre>
            <genre>Drama</genre>
        </genres>
        <year>1990</year>
    </movie>
</movie_collection>
```

```json
[
  {
    "title": "2001: A Space Odyssey",
    "director": "Stanley Kubrick",
    "actors": [
      "Keir Dullea",
      "Gary Lockwood",
      ...
    ],
    "genres": [
      "Sci-Fi",
      "Adventure"
    ],
    "year": 1968
  },
  {
    "title": "Goodfellas",
    "director": "Martin Scorsese",
    "actors": [
      "Ray Liotta",
      "Robert De Niro",
      "Joe Pesci",
      ...
    ],
    "genres": [
      "Crime",
      "Drama"
    ],
    "year": 1990
  }
]
```

Figure 4.2: Files containing information for 2 movies, XML (left) and JSON (right)

5. Case introduction

5.1 Company Background

5.1.1 Aveso

This thesis has been done in collaboration with Aveso Oy, which is also my current place of employment. Aveso is an IT company based in Turku, currently employing roughly 20 consultants in data management and integrations. Since its foundation in 2014, the company has released several software solutions and worked with customers in telecom, aviation, finance, and manufacturing on numerous projects. Aveso Data Studio is their most recent product, which controls master data and data quality processes. Furthermore, as Aveso is a certified partner with IFS, many business cases revolve around installing, upgrading, and managing IFS software for customers.

5.1.2 IFS

IFS is an international software company that develops and delivers enterprise software resources. The company was founded in 1983 in Sweden and has since expanded to offer its suite of products, IFS Applications, worldwide. Some of the main elements of the IFS Application suite are *enterprise resource planning* (*ERP*), used to manage business processes, and *enterprise asset management* (EAM) solutions. The newest version of IFS is IFS Applications 10, employing the new Aurena user interface. As of 2021, IFS has 4,000 employees, 10,000+ customers and 1,000,000+ users of its solutions [33].

5.2 Current Integration projects

Many of the integration projects developed by Aveso specifically revolve around IFS ERP solutions. Traditionally, ERP data have been retrieved and added to IFS using PL/SQL package procedures. Datasets are collected from one location, such as XML files inside a folder on a customer's server, then parsed, processed, and transferred to their IFS system. Many integration services are installed on-premises on the same host as the IFS ERP. Lately, Aveso has been investigating new ways to modernize its integration projects. An iPaaS-based solution would allow for the use of state-of-the-art features and could seem more enticing to customers. A cloud service that has often been mentioned in discussions with clients is Microsoft Azure, specifically its integration platform, Logic Apps.

5.3 Azure Logic Apps

Azure Logic Apps is Microsoft's iPaaS solution, first released in 2016 as a part of Azure's integration suite. The platform is used to integrate both on-premises and cloud services and had a reported user base of 40,000 as of September 2020. Microsoft has offered regular service updates ever since, such as support for Visual Studio Code and increased hosting alternatives [34].

5.3.1 Logic App Workflow

Logic Apps Designer offers a visual approach to creating integration workflows without writing code. Every process is based on a trigger, followed by a series of steps containing processes or tasks. In Figure 5.1, an example workflow is displayed.

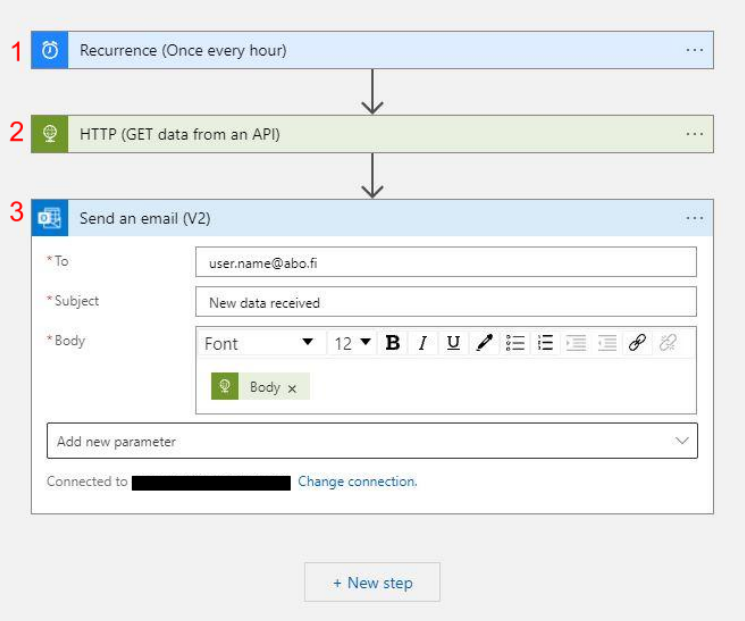

Figure 5.1: Basic Logic App workflow that runs once every hour, fetching data from an API and sending them to a specific email address

The first step of a *logic app* is the *trigger* (1), the condition that prompts the remaining *actions* (2,3) to run. Actions are separated into two types; *Built-in operations* (2) are run natively inside the logic app and are therefore not associated with any specific service. Examples of this are conditional operations, the creation of variables, and HTTP requests. Conversely, *managed* (further separated into standard/enterprise) *connectors* (3) allow creators to access apps and systems separate from the workflow process. Many of Azure's services and Microsoft's products can be accessed through these connectors, in addition to supporting actions for programs such as Oracle DB and IBM. For example, Logic apps can

detect emails from Office Outlook or read files from an on-premises file system using data gateways. Lastly, Microsoft allows the community to create custom connectors for their logic apps [35] [36].

5.3.2 Pricing

Azure Logic Apps offers numerous pricing options. When using the standard plan, users are billed by the hour, separately for CPU and memory usage. Another option is the consumption plan, where the user is billed based on the triggers and actions employed inside the logic app workflow. The price per execution is separate for actions, standard connectors, and enterprise connectors, the latter being the most expensive. When using the consumption plan, there is also a data retention fee. Lastly, a dedicated integration service environment can be set up to secure connections to applications in addition to added features. The dedicated service environment is the most expensive option by a significant margin, even when selecting the cheapest alternative out of the two available service environments.

An integration account must also be created to access specific capabilities, such as business-to-business connectors. Microsoft offers a pricing calculator on their website to help with selecting the appropriate logic app plan and features for users' projects. By estimating the number of connector executions per day in addition to integration service environments and integration accounts, upfront and monthly costs can be projected. [37]

5.4 The Case

One notable feature introduced in IFS Applications 10 is the support for API calls, meaning that ERP data management is no longer exclusively done via PL/SQL package procedures. Thus, Aveso has decided to use API calls in some of its integrations. The change raises the question of how Aveso's current IFS ERP integration services would have to be modified to allow these HTTP requests. Implementing the new feature would require modifications to code to run using existing integration service logic. Considering that changes are inevitable, now would also be an appropriate time to investigate a possible cloud migration of the service. Azure Logic Apps has been deemed a prime subject for potential migration. Before any actual migration is done, it is essential to ensure that Logic Apps meets the needs and demands of Aveso's projects. Cloudstep has been chosen as a decision process. Used to examine factors such as cost and performance and

identify constraints, Cloudstep allows for a well-informed decision regarding whether cloud migration is beneficial for the legacy application.

The subject of this thesis is to use Cloudstep to investigate the suitability of a migration to Logic Apps by creating profiles of Aveso and Azure to identify constraints. A small-scale "pilot" project is carried out when the process is finished, as suggested in the Cloudstep paper. After following all the steps of the decision process, the suitability of Azure will be assessed. Moreover, the Cloudstep process itself will be examined, analyzing its strengths and weaknesses. Any suggestions for improvements to the model will be listed. If any pitfalls are detected, they will be listed so that others facing similar migration decisions can avoid them in the future.

6. Cloudstep Decision Process

This chapter will apply process activities defined in the Cloudstep paper [15] to the Azure Logic Apps migration. The paper contains several illustrative questions that can be answered for each phase to assist readers. Many of these questions will be used as a guideline in the forthcoming steps. Although Cloudstep stages are described only once in this chapter, many have been iterated several times, notably the application profile. The description and creation of multiple profiles in separate iterations have been omitted. Changes caused by constraints are instead pointed out when relevant for each step.

6.1 Define Organization Profile

Legal and administrative attributes are detailed in the organization profile. First, it is vital to determine why the company is considering migrating the cloud. A large part of the driving force behind a cloud migration is the increased popularity of cloud services, which has led to competitive prices and perpetual development in the form of optimizations and new features. Aveso staff has personally heard numerous customers mention that Azure is their current integration platform of choice, making Logic Apps relevant for further research. Aveso's current integration services are only installed on-premises, which some customers can perceive as legacy software. In a progressively more competitive market, it is essential to offer up-to-date solutions in order to entice new business partners. As Azure is a well-established platform in the industry, expanding Aveso's integration service portfolio to include Microsoft-hosted options would benefit the organization by attracting many possible customers.

Aveso's computing resources are divided between servers managed on-premises and deployments on cloud platforms. Likewise, there is no uniform method for developing, testing, and deploying services and software products. Typical projects are, however, generally developed locally before publishing them on a customer's remote server or cloud service. It is worth noting that several of Aveso's projects are either published on Azure or use Azure's tools. This means that while the organization's IT professionals have no experience with Logic Apps integrations, they are nevertheless familiar with the Azure ecosystem and many of its features. In addition, Aveso's personnel are experienced in the field of data integrations in general, meaning they most likely possess the proficiency needed to develop and

maintain workflows using a new platform. Monthly meetings are held in the company to review the staff's competence regarding new trends in the business. If shortcomings are found in an area, a team member will research the subject. The moderate staff size of Aveso means that large-scale projects must be planned well in advance to reduce the risk of the company becoming shorthanded. One of the questions brought up in the Cloudstep paper is if the physical location of the organization's data and applications are in some way restricted by law. No evidence of any legal restrictions has been found specifically on the physical aspect of data storage. However, the *General Data Protection Regulation* (GDPR) rules undoubtedly apply to Aveso. Physical restrictions could potentially be determined in project contracts between companies on a case-to-case basis.

6.2 Evaluate Organizational Constraints

Based on the organizational profile that has been created, no organizational constraints have become apparent at this stage. Potential restrictions listed in the Cloudstep paper, such as IT staff fearing dismissal or reduced governance over IT resources, can widely be ignored as Azure cloud services are already ingrained in Aveso's current solutions. The risk of unauthorized access to data stored on the cloud is a topic without a definite answer. It can be debated whether a global technology leader's security system is more secure than the server room of a commercial building. The accessibility of data from outside the organization can place some constraints on the possible cloud-based alternative. As legacy applications are installed on-premises in the same location as the input folder and IFS installation, their way of accessing the ERP is straightforward. For a cloud service to access the IFS ERP not located in the same endpoint, the complexity of the necessary changes depends on the server's configuration. Firewalls and security certificates are areas where modifications can be required.

6.3 Define Application Profile (Aveso Integration Framework)

The subject of analysis is the batch integration interface Aveso Integration Framework (***AvIF***). *AvIF* interfaces work similarly to Windows Services and are run as background jobs. Many of Aveso's IFS ERP projects have used this method of deploying and running code. The application profile is split into two sub-activities: Usage- and technical characteristics.

6.3.1 Usage Characteristics

The main feature of the AvIF application in question is to read XML files from a folder, parsing their contents and adding the data to the customer's IFS ERP system. Data is modified according to a mapping scheme that has been created per the client's requirements. Some values are reformatted or concatenated, while others are fetched from separate mapping tables to match predefined values. The integration service creates part catalogs, inventory parts, purchase parts, and sales parts, all using values from the same XML file. By modifying the *config* file in the service's install directory, users can set the file paths from where XML files are retrieved and processed and where invalid files are moved. The AvIF interface runs on a set time interval, which can also be modified in the config file. The user can select the interval in seconds, minutes, or hours, along with the possibility to run the service at a specific time of the day. Several AvIF-interfaces can be deployed on the same location to handle separate tasks. Services are controlled via an interface called AvIF Manager, where they can be started, stopped, and reloaded. Additionally, AvIF Manager can view logs generated by the services. Logs created by AvIF are also saved in Windows' Event Viewer.

As the integration services are installed on the same physical location as the customer's IFS ERP, they can only be accessed remotely, thus requiring the creation of Windows users specifically for integration admins. By virtue of being a windows service, the AvIF integration does not require any interactions by the user once the service has been started in AvIF Manager. The integration service's usage patterns depend on the selected poll interval, albeit it is executed consistently. Most ERP handling is not time-critical, and therefore scheduling intervals below an hour are uncommon. While AvIF runs 24/7, most data are handled during office hours when people create parts. The cost to operate and maintain AvIF is comparatively low because the services are installed on preexisting hardware hosting the IFS ERP. Slight modifications to the code can be done quickly, although changes to the input XML file structure can demand rewrites of specifications and mapping tables. The integration is expected to handle between 10 and 100 XML files each day.

6.3.2 Technical Characteristics

All AvIF interfaces are written in C# and are developed specifically for Windows hardware. The current AvIF integrations are running on a Windows Server operating system. No release version has been targeted explicitly, meaning that the interface can also be run on various Windows operating systems. No minimum hardware configurations have been specified, although a minimum RAM size of

4GB is recommended, a limitation that should not be an issue for modern servers. Interfaces are coded and built internally by Aveso before deploying them to the customer's environment, as are any subsequent updates created for the service. The interface is limited to the server's private computer network, called *intranet*, and accessing the service from another location requires a Remote Desktop connection.

There are two separate AvIF services installed on the customer's server: part handling and product data management, the former being the only one investigated for cloud migration for now. A simplified architecture model for the integration interface can be seen in Figure 6.1. The part handling service loops through all XML files found inside the input folder and looks for the root node called **<objectArray>** for each one. If the node exists, there are data in the file, and the service can proceed to the next step. Now, each <**item**> node is parsed separately, validating the contents and then modifying them to match the expected formats for IFS. Part catalogs, inventory parts, purchase parts, and sales parts are either created or updated in IFS in the specified order. Some steps, such as purchase part handling, can sometimes be skipped according to rules in the mapping table row for the inventory part type. Each instance is handled in a separate C# method where data are transformed and finally sent to the IFS ERP via an SQL call. The AvIF interface that is being migrated uses .NET version 4.5. The .NET framework is utilized to connect to the IFS ERP and .NET runtime to execute the service itself. Concerning data handling, the service uses the LINQ interface to handle XML files. The IFS ERP is an Oracle database, and SQL calls are done using *Oracle.ManagedDataAccess* framework drivers.

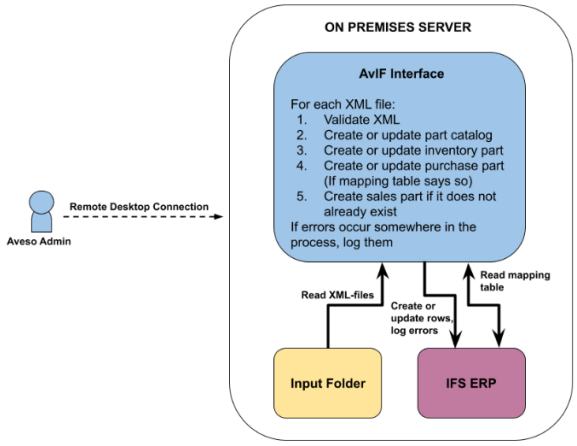

Figure 6.1: AvIF Solution Architecture

Regarding quality-of-service requirements, the service is expected to run continuously, executing systematically according to the predetermined time

interval. If any errors occur during the XML validation or data transformations, it is logged for the Windows Event Viewer. Additionally, a row is added to an error-logging table created specifically in IFS for the service.

6.4 Define Cloud Provider Profile (Azure Logic Apps)

6.4.1 Main Features

As Logic Apps offers many features and services due to its interconnected nature with other Azure products, the scope has been limited to only present features relevant to the solution. Logic Apps is just a part of the Azure Integration Services suite, with Azure Functions being the sole additional component used in our integration case. Microsoft's iPaaS-solution allows users to run logic applications locally and on-premises in addition to on Azure. Aside from the visual Logic Apps Designer tool, processes can be developed using Visual Studio Code on Windows, Linux, and macOS. Estimating the total price for the cloud-based integration can be difficult due to the potential use of several Azure services with separate costs. For the logic app, the consumption pricing plan was selected for the current integration as it offers the best value for the project scope. It is expected that the total executions for one month should equal a price below 10 euros when using the prices per execution listed in Table 6.1.

Table 6.1: Consumption plan pricing as of October 2021 [37]

	Price Per Execution
Actions	0,000022€, First 4 000 actions free
Standard Connector	0,000108€
Enterprise Connector	0,000857€

6.4.2 Solution Architecture

In Figure 6.2, the architecture of an Azure iPaaS solution for part handling is presented. ***The logic app*** is where the main application logic and workflow are located. The logic app is where the code from the legacy integration must be replicated using the Logic Apps Designer. Many aspects of the original AvIF service can be reproduced using standard actions, such as string concatenation. Some operations, notably API calls to an on-premises ERP and file reading, require more advanced methods. An on-premises data gateway is combined with a File System connector as a trigger for the logic apps workflow to detect and read files from a folder on a server. Microsoft's cloud services can access data that would

otherwise be unreachable externally by installing an ***on-premises data gateway*** on the server where the input folder is located. The gateway is run in the background as a Windows service and allows multiple users to connect to the same data source.

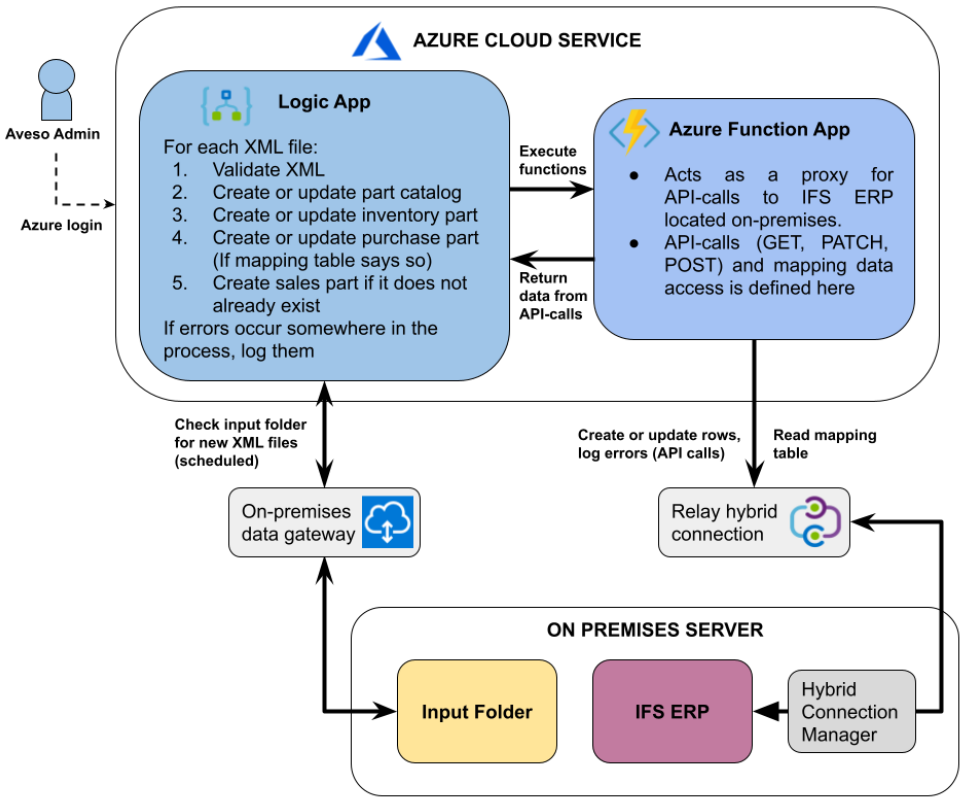

Figure 6.2: Azure Logic Apps solution architecture. NB: The input folder does not have to be situated in the same location as IFS ERP

A *proxy server* can be required (see 6.5 and 6.6) when API calls are done from a logic app to the IFS ERP located on-premises. The proxy is realized by creating a ***Function App*** for our service, containing ***Functions*** corresponding to each API call to the ERP system. Azure functions are used to build serverless apps by triggering user-created code based on events in Azure or third-party services. Functions can be created locally, and Microsoft offers support for many languages, for instance, C#, Python, and JavaScript. In this case, the functions were written in C# using Visual Studio's built-in Azure Function feature. Inside the Functions, API calls are executed using ASP.NET's *HttpClient* package. Once the functions are published to a Function App in Azure, they can be accessed inside a logic app workflow. From the logic app's perspective, the function is selected along with its input parameters. All API calls occur inside the Functions, allowing the logic app to receive the function's result data and proceed to the next step. In order to access

on-premises services, such as the IFS ERP's API, the Function App must use a ***Relay Hybrid Connection***.

Azure Relays allow services in a network to be visible to the cloud without opening any ports in a firewall. Hybrid connections are a new feature to the relay services, offering the possibility to send requests and receive responses using HTTP and WebSocket protocols. For a hybrid connection to work, a relay agent must be deployed to a location where both Azure and the needed endpoint can be contacted. In this case, the ***Hybrid Connection Manager*** (HCM) is installed to the on-premises server, working as the relay agent. HCM connects to Azure Relay using port 443, while the application (in our case, the proxy Function App) also connects to the relay. TLS 1.2 is employed for connection security, whereas shared access signature keys are utilized for authorization and authentication. Function Apps are billed separately from logic apps. Hybrid Connections are only available in the premium plan, to which prices can be seen in Table 6.2. There are no execution charges in the premium plan, and the user instead pays based on memory allocation and the number of core seconds. The function app can scale its instances according to load, but at least one instance is allocated on any occasion. If a Function App proxy is required for the logic app, the prices of actions and connectors found inside the logic app become insignificant compared to the monthly fee of over 100€ that is paid for the premium Function plan.

Table 6.2: Premium plan pricing for Function Apps as of October 2021 [38]

Meter	Price
vCPU duration	105.131080€ vCPU/month
Memory duration	7.491377€ GB/month

6.4.3 Security, Support, and Logging

When a logic app is created, the deployment region can be selected out of the 20 regions around the world where Azure operates. The closest options for Finland are North and West Europe, with data centers in Ireland and the Netherlands, respectively. Microsoft's SLA guarantees the availability of logic apps to be 99.9%. If the Service Levels are not met, customers will be granted Service Credits used to pay monthly fees. Azure's infrastructure complies with pivotal reliability and security standards *NIST SP 800-53* and *ISO/IEC 27001:2013* [39]. Microsoft currently employs over 3,500 security experts and places considerable investments

into cybersecurity research. Some examples of steps taken by Microsoft to increase security are the isolation of Microsoft- and customer networks for protection against attacks and built-in mechanisms to guard services against distributed denial-of-service (**DDoS**) attacks. The person or organization responsible for managing the security of the application service varies depending on what features are activated in the Azure subscription. For instance, Azure Defender is a tool integrated with Security Center that offers threat protection for Azure workloads. Web Application Firewall and Application Insights are other services that can be selected for Azure,

Alternatively, the logic app can be added to an *Integration Service Environment* (*ISE*) to isolate it using an Azure virtual network. The pricing of 0.97€-6.02€/hour makes this feature outside the current integration budget. Application Insights can furthermore be used to analyze the executions and logs of Logic Apps and Functions. If users want to examine earlier logic app runs, they can be viewed in the workflow designer window. The executions list displays the execution status of the latest logic apps runs if the user only wishes for a quick overview of process executions.

6.5 Evaluate Technical and Financial Constraints

The most apparent restrictions in the Azure Logic Apps integration involve **communication constraints**, specifically the lack of support for self-signed certificates in HTTP requests. For customers whose servers are using SSL certificates, this is a non-issue, albeit, for others, this necessitates the purchase of a certificate, the use of a proxy, or the use of an integration service environment to circumvent the restriction. These options tie into the next area of constraints, **financial**, which are therefore highly dependent on the customer's existing server infrastructure. Although they can be relatively inexpensive, it can be seen as unreasonable to expect the customer to purchase an SSL certificate for their server(s) for the sole reason of being compatible with Aveso's solution. Therefore, the approach involving certificates is omitted in this case. Using an ISE would add support for self-signed certificates, but for Aveso's clients, the monthly price increase of at least 700€ would be over the budget. This leaves some sort of proxy solution as the only viable option. To summarize: If the customer's servers use SSL certificates, the cost of an integration service using logic apps will be low, with a monthly cost in the range of 5-10€. For the rest of the customers, most of the integration's expenses will go to creating and maintaining a proxy.

Next up are **organizational constraints**. Aveso's staff are familiar with many of Azure's services and would undoubtedly have the skills necessary to manage a logic apps integration. Comparing Figures 4 and 5 demonstrates that the iPaaS-solution would undeniably be more complex, especially if a proxy system is used. Managing both a logic app and a proxy is a sizable endeavor compared to the comparably simple legacy service. For example, one small change in the IFS ERP API could require code adjustments and redeployment of the Azure Function App in addition to updating the Logic App. There are no glaring issues regarding **security constraints** when migrating the legacy application to Azure. Aveso already hosts several of its products via Azure, and there have not been any problems concerning the security of customers' data compared to on-premises solutions. Nevertheless, this aspect should be discussed with clients in advance to verify that they approve of Azure's security measures. **Performance constraints** are not as critical to the integration service as **availability constraints**. As long as the service is run consistently, processing speeds have minimal impact. The only requirement is that the integration process must finish its execution before the next scheduled run is started. Azure's SLA of 99.9% availability, corresponding to maximum downtime of 1m 26s each day, makes it highly unlikely that a process run once or twice every hour is impacted. What is more, if one scheduled run is skipped due to Azure downtime, the XML files can be handled in the subsequent execution. No problems were discovered when evaluating **suitability constraints**, that is, changes needed for the application to make it fit the migration.

6.6 Addressing Constraints and Assessing Other Cloud Providers

6.6.1 Address Application Constraints

There are no severe constraints that require a change in migration scope. In our case, the issue involving self-signed certificates in API calls will be handled using Azure Functions and a Hybrid Connection. The Azure Functions are coded in C#, resembling a legacy application. All HTTP requests used in the integration are executed through the methods defined in these functions. Using a proxy depends on the client's infrastructure, so financial constraints are highly variable. There is no decisive way to address these constraints, and they should instead be evaluated on a case-to-case basis. For the migration in this thesis, using Function Apps is required, and its operating costs are within the budget for the project. Organizational constraints can be disregarded since Aveso's staff have the

competence required to create and manage both the Logic App and the Function App. There are no critical constraints that stop us from creating a pilot project at this stage.

6.6.2 Change Cloud Provider

As there are constraints regarding Azure, other cloud services should be inspected to solve the constraints defined earlier. Investigating an alternate cloud provider is redundant in our case, as the only feasible option is Azure. Instead, we have considered how the solution defined in Figure 5 could be modified to benefit different cases. First, suppose the customer's IFS ERP endpoint uses an SSL certificate. In that case, the proxy can be disregarded from the solution, and HTTP requests can be made directly via the logic apps workflow using standard connectors. This solution is ideal, offering the cheapest operating cost and most straightforward management as only the logic app and on-premises gateway would have to be maintained. As stated before, the method above is ignored, as we cannot assume that all clients have purchased certificates. An ISE could be used if the integration was done for a large-scale project and budget was not a concern. Even though the operating costs would skyrocket, the need for proxies would be eliminated. There would also be the added benefits of the dedicated environment, namely isolated storage and more reliable performance.

If a proxy is necessary for the integration, it could be created via some third-party software or service instead of Azure. These might be more cost-efficient than using Function Apps, but it was decided that it would be best to keep the solution contained in the Azure environment. Functions are highly compatible with Logic Apps and can be accessed from the workflow designer straightforwardly. Another modification could likewise be done if Functions are used; much of the logic inside the logic app workflow could be moved inside the functions. Compared to C#, some data operations are cumbersome to code in the Logic Apps Designer. If/else-checks and complex string manipulations are some actions that would be cleaner to write inside the Functions. This modification would have little monetary benefits, as the price of individual actions in the logic app is minimal. Additionally, the logging potential of the process would be reduced. When viewing logic app executions of the workflow, the results of individual actions and branch decisions can be discerned. To achieve the same level of logging inside of functions, in-depth error handling and application insights must be used. This function-based method is a good approach but moving any extra logic to the functions was avoided for the pilot case.

6.7 Define Migration Strategy

Considering the 6 R's defined in 3.2, the migration strategy used in this case mainly involves Repurchasing. Instead of using legacy software developed and deployed by Aveso, the integration service will be moved to a new product, Azure Logic Apps. There are no convenient ready-made services available on Azure, so the logic app will have to be developed manually by Aveso. The original code, written in C#, will be translated to the visual Logic Apps workflow view while maintaining all its features. To some extent, the migration resembles re-architecting because all the features cannot be moved straightforwardly to Azure. Several aspects, such as the change from SQL queries to API calls and the need for a proxy, indicate that the solution must be reworked. The technical documentation used for the original on-premises service containing mapping rules and administrative information will be used as a guideline when creating the logic apps workflow (see Appendix B). The next chapter will discuss how these guidelines are transformed in logic apps where the pilot project is created.

6.8 Pilot Project

No actual migration will be performed in this thesis. In the Cloudstep paper [15], Beserra et al. suggest that while defining the migration strategy, organizations should start with a scaled-down "pilot" project to ensure that the cloud service works as intended. The part handling AvIF integration will be used as a template for the Logic App pilot project. The scope is reduced, and only the handling of part catalog and inventory parts are replicated. However, the logic for handling the remaining components, purchase- and sales parts, correspond very closely to what is handled in this small-scale test. They do not contain any unique behavior not seen anywhere else in the code. Therefore, it can be assumed that if the processing of part catalogs and inventory parts functions correctly, so would the rest of the legacy application.

The pilot project's creation will be presented in this step, first focusing on the general parts of establishing a logic app, followed by in-depth examinations of how the fundamental aspects of the AvIF integration are handled. Results and analysis are discussed in the next chapter. Due to the visual nature of Logic Apps, there will be plenty of screen captures of the workflow and its settings. These can also be useful for anyone attempting a logic app migration in the future.

6.8.1 Essential Functionality

The following functionality has been established as essential in the AvIF solution. Consequently, Azure must be able to handle the following tasks to replace the existing application:

1. Support various poll modes, either specific times or intervals.
2. Detect and read XML files from a folder located on an on-premises server.
3. Validate that the XML file is in the correct format.
4. Map/reformat XML data using a set of rules.
5. Fetch additional data from a mapping table if required
6. Create and update elements in IFS ERP based on XML data.
7. Log and handle errors if they occur.

6.8.2 Creating a logic app

Logic Apps are created by going to the Logic Apps resource list in Azure and selecting “Add”. In order to be able to create Azure resources, a Subscription and Resource Group has to be created. The Azure subscription is used for management and billing reasons, supporting several payment methods. Subscriptions allow for creating resources that are then grouped into logical collections. These resource groups can contain any Azure assets, such as databases, apps, and virtual machines. Figure 6.3 is a screenshot that displays the settings for the current logic app. Aveso is a member of the Microsoft Partner Network, which grants several benefits, including the use of Microsoft’s unique subscription method. The existing *AvesoDev* resource group was selected, and the consumption type (pay-as-you-go) was chosen for the app. Once the resource is deployed, it can be accessed from the list of Logic apps. When the app is selected for the first time, Azure offers a wide range of templates and triggers as a starting point for the workflow. In our case, the default blank Logic App will be used.

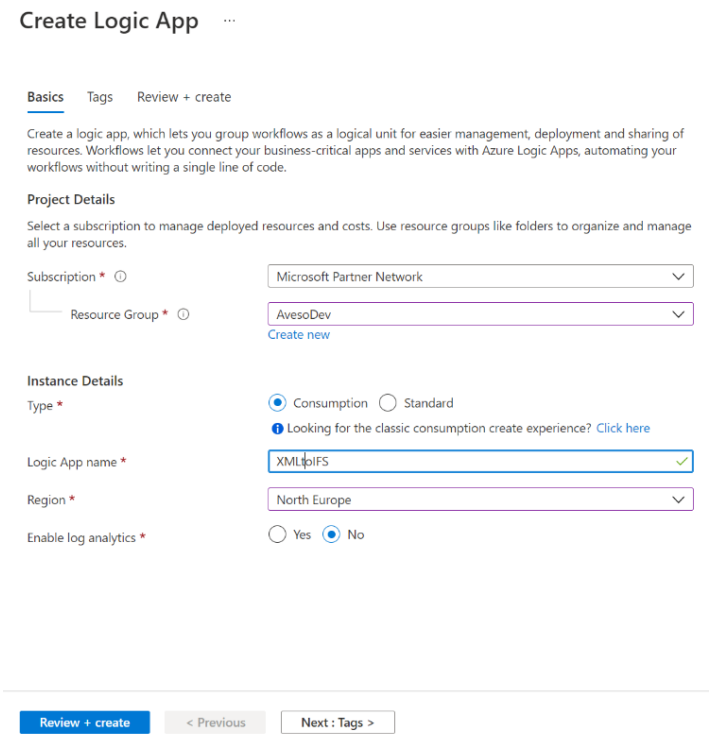

Figure 6.3: Create Logic App window

6.8.3 Selecting a Trigger

Selecting the blank Logic App template opens the workflow designer view. Every app has to start with a trigger. Out of all the triggers available in Logic Apps, the "File System – When a file is created" is the most appropriate. The user can choose how often the trigger checks for items in the unit "polls per time unit", where the selectable time units range from seconds to months. The trigger selected for this app inspects the folder two times every hour, equaling every thirty minutes. The trigger and its settings are visible in Figure 6.4 (left). The XML files will be read from my PC, and I have created the folder *LogicAppFolder* on my C drive specifically for this application. The File System connector cannot automatically access the computer; hence an on-premises gateway must be created between my PC and the logic app.

6.8.4 Connecting to an On-Premises Folder with a Gateway

An on-premises data gateway is installed by launching the installer on the endpoint with access to the folder used to trigger the app. In this case, the gateway was installed on my PC. The gateway is linked to the user's Azure account during the installation process, and it is given a unique name. After the gateway is registered, it must likewise be linked to a gateway resource in Azure. Users select amongst all installed gateways for the chosen resource group when creating a gateway resource.

Now the gateway can be accessed from connectors in Logic Apps. In Figure 6.4 below, the file system trigger uses the *Marcus-PCLogicApps* gateway, which in turn is linked to the *MarcusLogicApp* resource found in Azure.

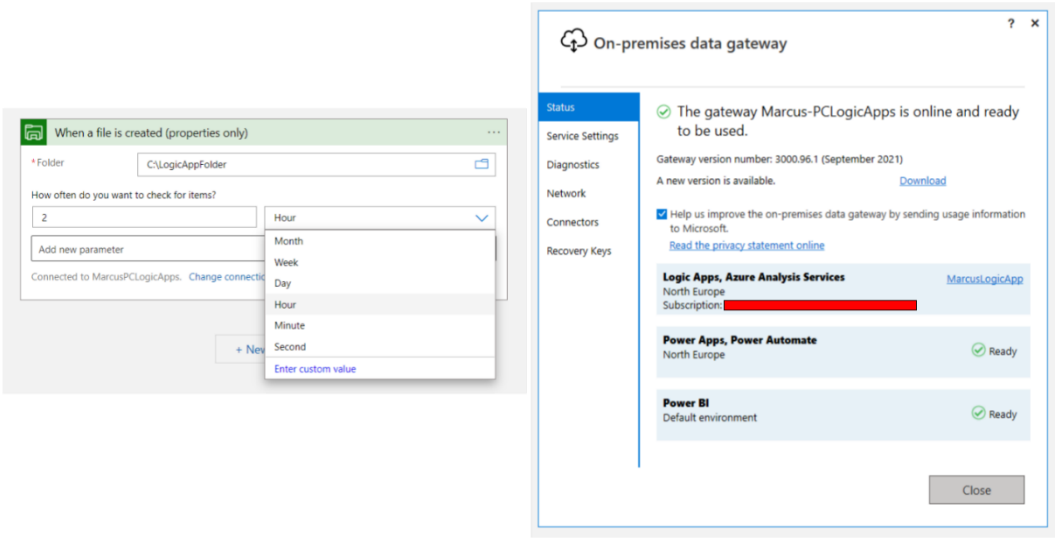

Figure 6.4: File System trigger connected to on-premises folder C:\LogicAppFolder (left). View of installed On-premises data gateway from my PC (right)

6.8.5 Looping Through and Validating XML Files

When a new file is detected in the specified folder, the next step is to use the "List files in folder" connector. Logic apps save the results of actions in various formats, allowing for dynamic content to be accessed later in the workflow. In our case, the "list connector" result's body, containing all files in the folder, can be referenced in a for-each loop. This loop makes it possible to process each file separately. Inside the loop, the first action is to retrieve the data from the file. Access is done by using the "Get file content" connector and the file path for each file in the folder. A screenshot of the finished process can be seen in Figure 6.5.

Before proceeding to any data mapping, the XML file should be validated to assure that it is in a correct format and contains all required nodes. XML validation is performed using a connector with the same name that takes a file's content and XML schema as inputs. Currently, the only way to create schemas is by adding them to an integration account. These Azure resources manage integration artifacts, including maps, certificates, and schemas. Several pricing tiers are available for integration accounts, including a free option. The free account offers a minimal number of integration resources, but fortunately, only one XML schema is necessary for our process. By opening a valid XML file in Visual Studio and

choosing “Generate Schema”, a schema was created and subsequently added to the account with the name *headerSchema*. All necessary inputs are now defined.

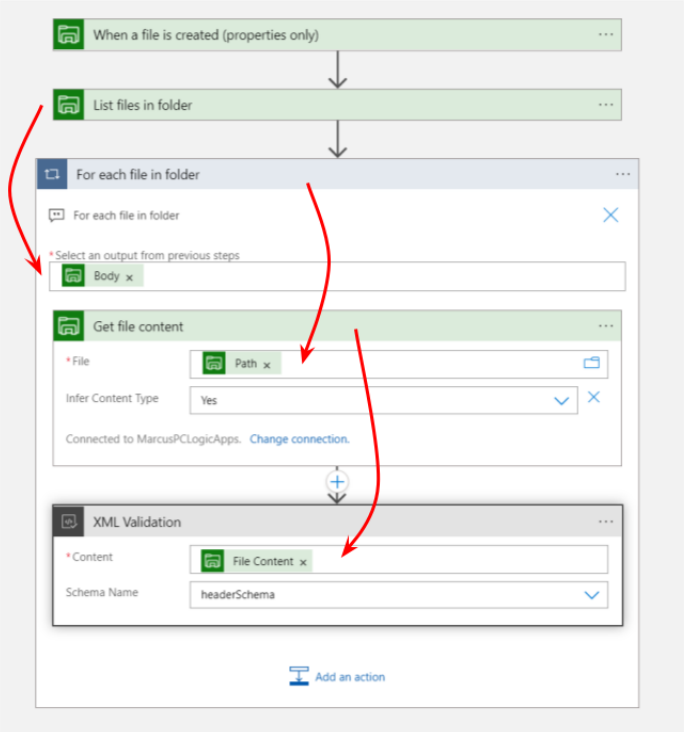

Figure 6.5 Looping and validation logic of XML files. Arrows have been added to visualize dynamic references

6.8.6 Mapping Data

Each XML file can contain several items inside of the *objectArray* node. The XML data is transformed to JSON arrays in order to conveniently reference the nodes in transformations and API calls. The XML-to-JSON conversion is achieved in the workflow by defining a schema that describes the structure of the desired JSON data. Now that the data are easily accessible, mapping can be performed. Mapping rules are taken directly from the documentation for the legacy application. A condensed version is found in Appendix Table 2. As seen in the mapping table, many different validations and transformations of varying complexity must be carried out. The most rudimentary verifications, such as ensuring that item status equals “ACCEPTED”, are best accomplished with an if/else-control action. All remaining workflow actions are placed inside the True case while remaining cases must be handled (see 6.8.9). Other mappings can be tackled using different methods. A switch control action was used to translate Finnish magnitudes (“kpl” and “mm”). If the magnitude is one of the Finnish options, the magnitude variable is updated to the corresponding English version. Otherwise, the magnitude is kept

in its original lowercased form. In Figure 6.6 below, the logic being discussed is displayed.

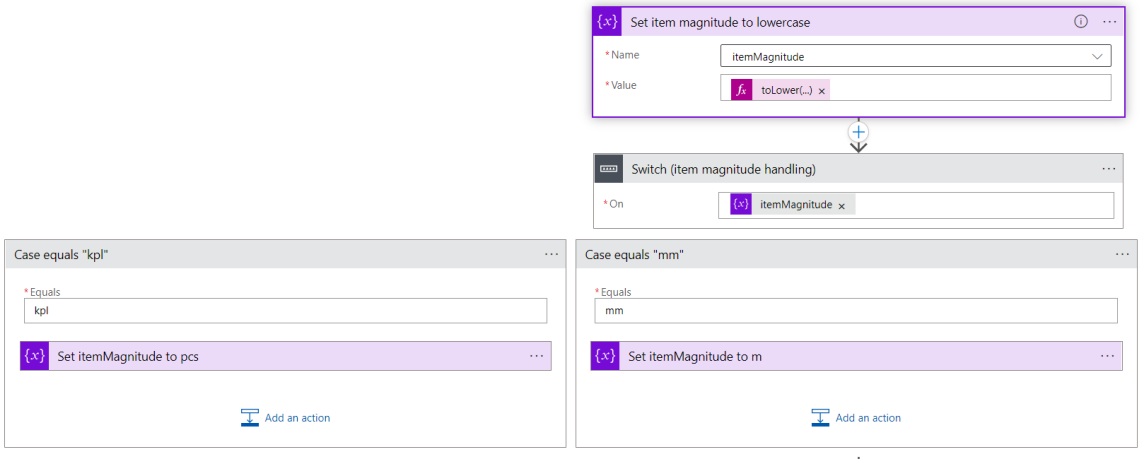

Figure 6.6: Item magnitude mapping using switch control actions. There are three switch cases: “kpl”, “mm” and default case (not visible in screenshot)

As seen in Figure 6.6 with the use of the *toLower* method, Logic Apps supports expressions in the form of string, logical, and conversion functions, amongst many others. Solving some of the mapping tasks using inline code can produce lengthy expressions. A case in point is the inventory part description field formed by combining the description_1- and 2 fields from the XML, separated with whitespace. The inline code looks like the following:

```
join(createArray(items('For_each_item_in_file')['item_desc1'], ' ',
items('For_each_item_in_file')['item_desc2']), '')
```

Other mappings are too complicated to handle purely with inline code. One of these is the *prime commodity* field for the inventory part, based on the *item_group* node in the XML. The field has a maximum length of five characters, where all leading letters are used, followed by at most two numbers. In the legacy application, this was solved using regex operations. In this instance, a JavaScript Code connector in Logic apps was utilized to achieve equivalent results.

6.8.7 API Calls and Proxy

Rows are inserted and updated in the IFS ERP using API calls. For this test project, the ERP is hosted on an Aveso server that uses a self-signed certificate. Therefore, it is necessary to create a proxy, which will be done with a Function App. First, a separate app service plan is created. The “Premium v2” plan, with a monthly price of 125,91€, is the least costly option that supports Hybrid Connections. Individual functions can be created once the Function App has been set up with the premium service plan. Visual Studio’s Azure Functions template was used to develop the

proxy. Listing 6.1 displays the code for the function used for Part Catalog PATCH calls in the IFS ERP API.

```
public static class PartCatalogPatch
    {
        [FunctionName("PartCatalogPatch")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "patch", Route = null)] HttpRequest req)
        {
            ApiClass.InitializeClient();

            var reader = new StreamReader(req.Body);
            reader.BaseStream.Seek(0, SeekOrigin.Begin);
            var rawJson = reader.ReadToEnd();

            PartCatalog part = JsonConvert.DeserializeObject<PartCatalog>(rawJson);

            bool success = ApiCalls.UpdatePartCatalog(part.PartNo, rawJson).Result;

            return new OkObjectResult(success ? "Success" : "Failed");
        }
    }
```

Listing 6.1: Part catalog PATCH function main class

The function takes a part catalog JSON object as its input parameters. The raw JSON is first extracted using the *StreamReader* class. Then, the JSON is deserialized to a *PartCatalog* object in order to reference its part number. It is sent to the "update" API call method with the raw JSON string (displayed in the next listing). The .NET *HttpClient* is also initialized via the *InitializeClient* function. Inside the function, default request headers are added to the client. The base URL for API calls and the username/password combination are added to the client as well, their values being defined in a locally stored JSON file. The full code, including the InitializeClient method, can be found in Appendix C.

In listing 6.2, the code for the API call itself is executed. The URL for the PATCH request is formed by the HttpClient's base URL concatenated with the relevant projection link, defined in IFS API documentation. The part catalog JSON string is added as content to the asynchronous PATCH call. If the request is successful and the part catalog is updated, the HTTP response is a "success" status code, and consequently, a Boolean true is returned from the function. GET, POST, and PATCH calls are defined for all part items using corresponding logic. The functions can be deployed directly to the Function App from Visual Studio.

```
public static async Task<bool> UpdatePartCatalog(string partNo, string partJson, ILogger log)
    {
        string url = "PartHandling.svc/PartCatalogSet(PartNo='" + partNo + "')";
        try
        {
            ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
            var content = new StringContent(partJson.ToString(), Encoding.UTF8, "application/json");
            ApiClass.ApiClient.DefaultRequestHeaders.Accept.Add(
                        new MediaTypeWithQualityHeaderValue("application/json")
            );

            using (HttpResponseMessage response =
                        await ApiClass.ApiClient.PatchAsync(ApiClass.ApiClient.BaseAddress + url, content))
            {

                if (response.IsSuccessStatusCode)
                {
                    return true;
                }

                else
                {
                    Console.WriteLine("Error");
                    throw new Exception(response.ReasonPhrase);
                }
            }
        }

        catch (Exception e)
        {
            if (e.Source != null)
                Console.WriteLine("Exception {0} source: {1}, {2}", e.InnerException, e.Message, e.HelpLink);
            throw;
        }
    }
```

Listing 6.2: Code for update part catalog API call

6.8.8 Hybrid Connection and Proxy Use in Logic App

The Function App is not yet able to connect to the on-premises server, as a hybrid connection must be set up in the same location as the IFS ERP. It is installed similar to the on-premises data gateway. When the hybrid connection is operating, it is linked to the Function App via the app's settings. The server's TCP port (48080) must be specified in the endpoint when adding the connection. Values stored in the local settings file (base URL, API username/password) must be added to the Function App's configuration settings manually in Azure. Finally, the public key certificate of the server hosting the IFS ERP needs to be included in the SSL settings of the Function App. At this stage, Functions can finally be used inside of Logic Apps. Figure 6.7 shows a view of the proxy call from the workflow designer. Values written in the request body field are passed to the proxy and, therefore, to the API. In this case, all values have been transformed to the intended format before calling the proxy.

6.8.9 Mapping Tables

Many XML fields, for instance, *item_type*, are used to fetch values from mapping tables in IFS. I did not have access to any of the tables employed in the legacy application. A custom projection was created and deployed in IFS for testing purposes, allowing it to access data via HTTP requests. The table contains only one row and two columns, functioning as inputs and outputs. In our primitive scenario,

the input of *value1* produces the output *value2*. There would be separate mapping tables containing more rows for each part type in genuine cases. Requests are handled through the Function App akin to API requests. Figure 6.8 shows a screenshot of a successful fetch from the custom IFS ERP mapping table using the *GetFromMappingTable1* function.

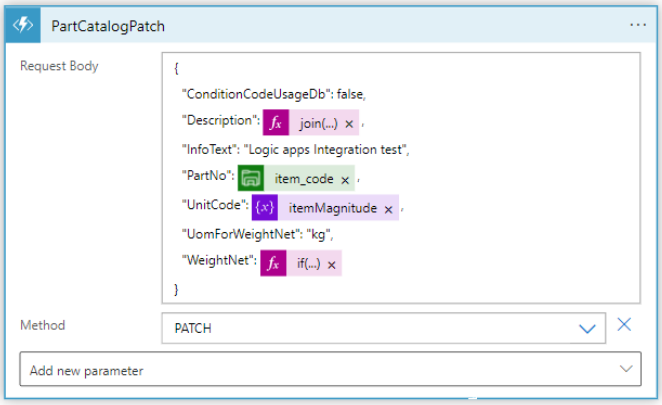

Figure 6.7: Part catalog PATCH functions. Parameters are a mix of hard coded and dynamic values. The request body values are sent to the Function App, which in turn makes the HTTP request using the parameters.

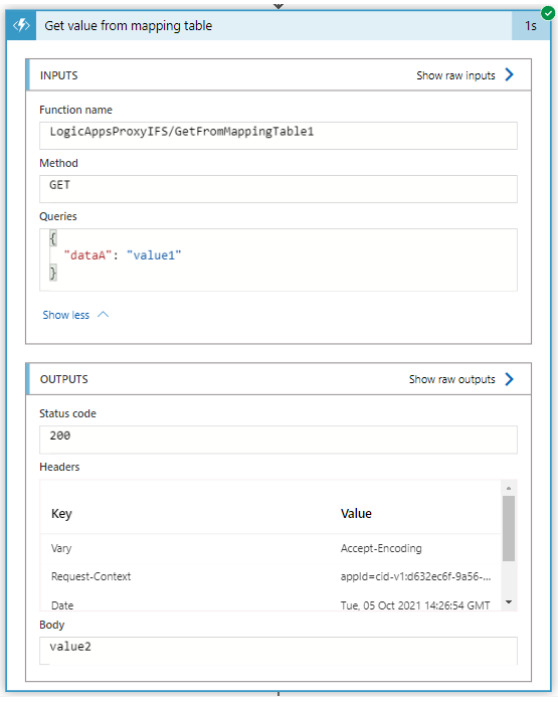

Figure 6.8: Successful request to a mapping table. “value1” is used as input, producing the output “value2”.

6.8.10 Error Handling

Error handling in Logic Apps is accomplished by a combination of *retry policies* and “*run after*” behavior. All HTTP requests, such as the one done through the

Function Apps proxy, have retry policies activated when connectivity exceptions occur or an HTTP status code signifying a failure is returned. The default policy retries the request up to 4 times with an exponential interval. The retry intervals can be tweaked, or the policy can be disabled altogether, but the default settings were kept in this instance.

If errors are to be handled for any other actions in the logic app workflow, they should be surrounded by a *scope*, followed by "run after" logic. These components closely match the try/catch-logic found in C# code. In the pilot project, all data mapping actions and API requests are moved inside a scope. The scope is followed by a control action that examines if the scope is completed successfully. By default, logic apps terminate their executions if an action fails. This practice can be altered by utilizing the "run after" settings. In Figure 6.9, the "run after" settings for the "If scope failed" control action is set so the workflow advances there even if the scope is timed out, skipped, or failed.

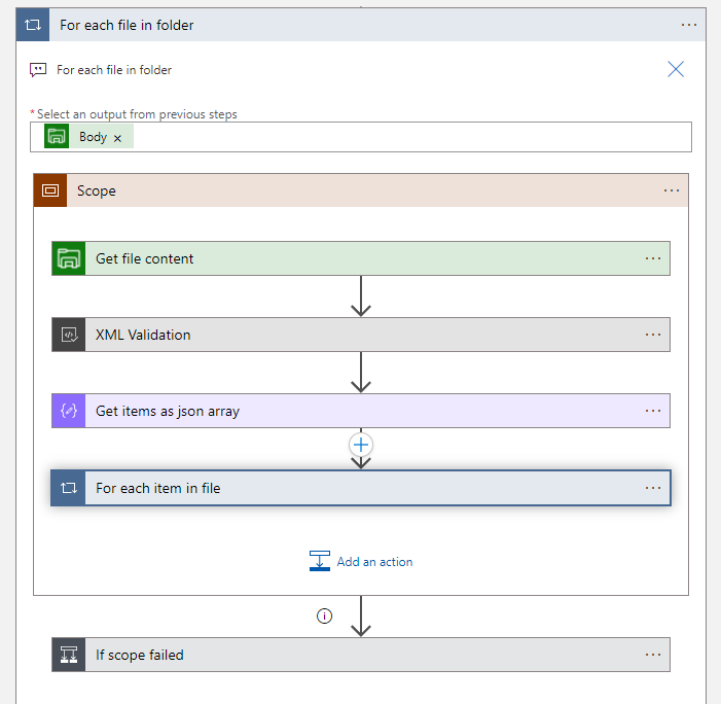

Figure 6.9: Mapping actions and API calls have been moved inside a scope. Most actions are located inside the "For each item in file" loop and are therefore not visible in this screenshot.

The control action checks if the scope's result status is either failed or aborted, in which case the error is to be handled. The AvIF application handled errors by logging them in a specific table in IFS through a SQL query. The same functionality is possible to produce with API calls, although no such table has been created in IFS for the test project in Azure. Until then, an empty function was created that is

called in error states. Relevant API calls can be placed inside of the function later. A notification could be sent to admins to look at the specific workflow execution for more information, as they can see precisely where the issue appeared using the visual tools. If all actions in the scope are completed without exceptions, the XML file is moved to a different folder, indicating that it has been handled. Figure 6.10 exhibits the "If scope failed" control action more closely.

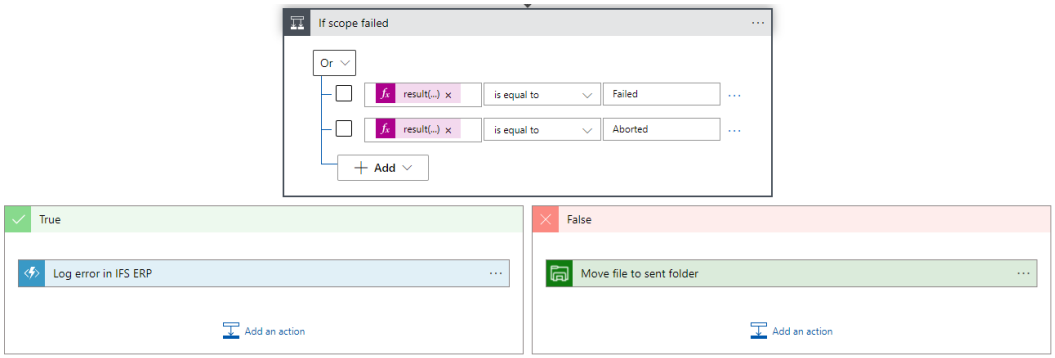

Figure 6.10: Error handling. There are different branches depending on if errors are found (True) or not (False)

6.8.11 Operating Costs of Pilot Project

The finished pilot project uses one trigger and 37 connectors. This sum includes the connectors in all possible branches in the workflow, and hence all connectors cannot be executed in one run. Estimating the operating costs for the app is challenging, as the service's usage can fluctuate daily. Because the logic app employs a consumption pricing plan, the cost for the actions and standard connectors can be added together to calculate the price. The maximum number of connectors that can be used in one execution is 29, of which 18 are actions and 11 standard connectors. This number assumes that the XML file only has one item, which is often not the case. By separating the operations inside the item loop(s) (10 and 7) from the rest in the workflow (8 and 4), the max number of actions (x) and standard connectors (y) used can be calculated with the equation:

$$x = 8 + \alpha * 10$$

$$y = 4 + \alpha * 7$$

where α = the number of items in the XML file. A reasonable number of items to loop for each XML file is 3. Inserting the value into the equation results in the following:

$$x = 8 + 3 * 10 = 38$$

$$y = 4 + 3 * 7 = 25$$

There can also be more than one XML file in the folder. Estimating that the total number of files handled per day is under 100, we can assume that the load is evenly distributed for simplicity. A poll interval of 30 minutes means that two files are handled each time the app is triggered. The average month contains 730 hours, meaning that the app is triggered 1460 times. We can assume that all operations are re-run for each file. Estimating the monthly cost for workflow operations with these assumptions equals

$$1460 * \beta\big((x * \ price\ of\ action) + (y * \ price\ of\ standard\ connector)\big)$$

where x = number of actions, y = number of standard connectors, and β = number of files. Inserting the values x = 38 actions, y = 25 connectors and β = 2 files each 30 minutes results in:

$$1460 * \ 2\ \big((38 * \ price\ of\ action) + (25 * \ price\ of\ standard\ connector)\big)$$

$$= \ 2920 * (38 * 0{,}000022€ + 25 * 0{,}000108€)$$

$$\sim 10{,}32€/month$$

The result does not consider that the first 4 000 workflow actions are without charge in the consumption pricing plan. A proxy is furthermore required for the solution. Using a Function App with a premium service plan V2 makes the estimated monthly cost for the pilot project:

Logic App operations (estimate)		10,32€
Function App service plan (case dependent)	+	125,91€
Total		136,23€

As expected, the operating costs for the Logic App pale in comparison to the Function App that serves as a proxy. It is also worth noting that estimations used in the Logic App price calculations (number of files and items) are "worst-case scenarios" with 100 files handled each day. This would not occur in practice, especially considering that these calculations include all weekdays, including Saturdays and Sundays, when there would otherwise be less activity.

7. Results and Discussion

In this chapter, the results will be analyzed. First, the outcome of the pilot migration project will be discussed, along with general impressions of Azure. Secondly, the benefits and drawbacks of using the Cloudstep decision process are examined. Finally, my personal impressions about the migration will be presented at the end of the chapter, with suggestions for further improvements.

7.1 Pilot Project

Overall, the pilot migration project was successful. All essential functionality of the legacy application was replicated on Azure, and the process was executed with anticipated results. The processing time of approximately 15 seconds was slightly larger than the legacy application, but the time difference can be neglected in this case due to the infrequent executions. The number of required operations, and therefore the price, was slightly higher than expected. Several caveats regarding a full-scale migration were exposed during the pilot project's creation. As such, it is best to discuss the positive and negative aspects of cloud migration separately.

7.1.1 Positive Aspects

The main advantage of developing Logic Apps processes is the visually oriented workflow designer. Compared to a C# application, starting with a new process is considerably faster when using the Logic App Designer. Utilizing templates in combination with actions and connectors provided by Azure reduces the level of coding experience required to create a functioning integration process. In Logic Apps, features such as threading can easily be enabled in for-loop settings, while logging is done automatically. Especially for integration processes of lower complexity, Logic Apps is a good option. Developing the proxy with Function Apps was also remarkably simple.

In contrast to testing and publishing code for AvIF services, Visual Studio's built-in support for Azure Functions made the process faster and more accessible. No longer do developers have to manually copy files to the existing installation via a combination of VPN and RD connections. Visual Studio allows for instant publishing of Functions to Function App by synchronizing the application with the developer's Azure account. Once configured correctly, the on-premises data gateway and hybrid connection work as intended and allow for increased flexibility

since the application is no longer tied to the IFS ERP's location, an integral restriction of the legacy application. Additionally, the input folder for XML files can be in a different location altogether, further increasing the possibilities for the integration.

7.1.2 Negative Aspects

As for the negative aspects of the pilot project, a noticeable issue is the limited support for data manipulations and operations in Logic Apps. This drawback results in a solution that appears overly complex, as it requires several operations for seemingly simple tasks. Listing 7.1 and Figure 7.1 display the same process in C# and Logic Apps, respectively. The code was created purely for demonstration purposes and contains data operations of similar intricacy to those found in the pilot project. The function examines the string variables *vehicle* and *price* and checks if the vehicle is a car that costs under the chosen *maxPrice* value. The result is then written to the *resultString* variable. Comparing the two, it is evident that the C# code is more concise than the logic apps workflow. Note the complexity of the workflow; While the logic app process makes logical sense and is seemingly intelligible if the workflow were zoomed in, placing several analogous chains of actions in a sequence would negatively impact the readability of the workflow. Additionally, inline code in logic apps workflows is placed inside expressions blocks that obscure much of their contents. The editor for writing expression code is also very cramped, consisting of only a single line. It is impossible to see the entire code for extended expressions in the editor, making it troublesome to write elaborate operations.

```
try
{
    switch (vehicle)
    {
        case "car":
            resultString = Int32.Parse(price) < Int32.Parse(maxPrice)
                ? "Price of car is " + (Int32.Parse(maxPrice) - Int32.Parse(price)) + " below max price " + maxPrice
                : "Price of car is too expensive";
            nrOfCars++;
            break;
        default:
            resultString = "Non-car vehicle " + vehicle + " is " +
                (Int32.Parse(price) > Int32.Parse(maxPrice) ? " over " : " less than ") + " the max price";
            break;
    }
}
catch (Exception e)
{
    //Handle error somehow
}
```

Listing 7.1: Code for checking the price of vehicles. The code uses switch-cases and integer conversions in order to compare values.

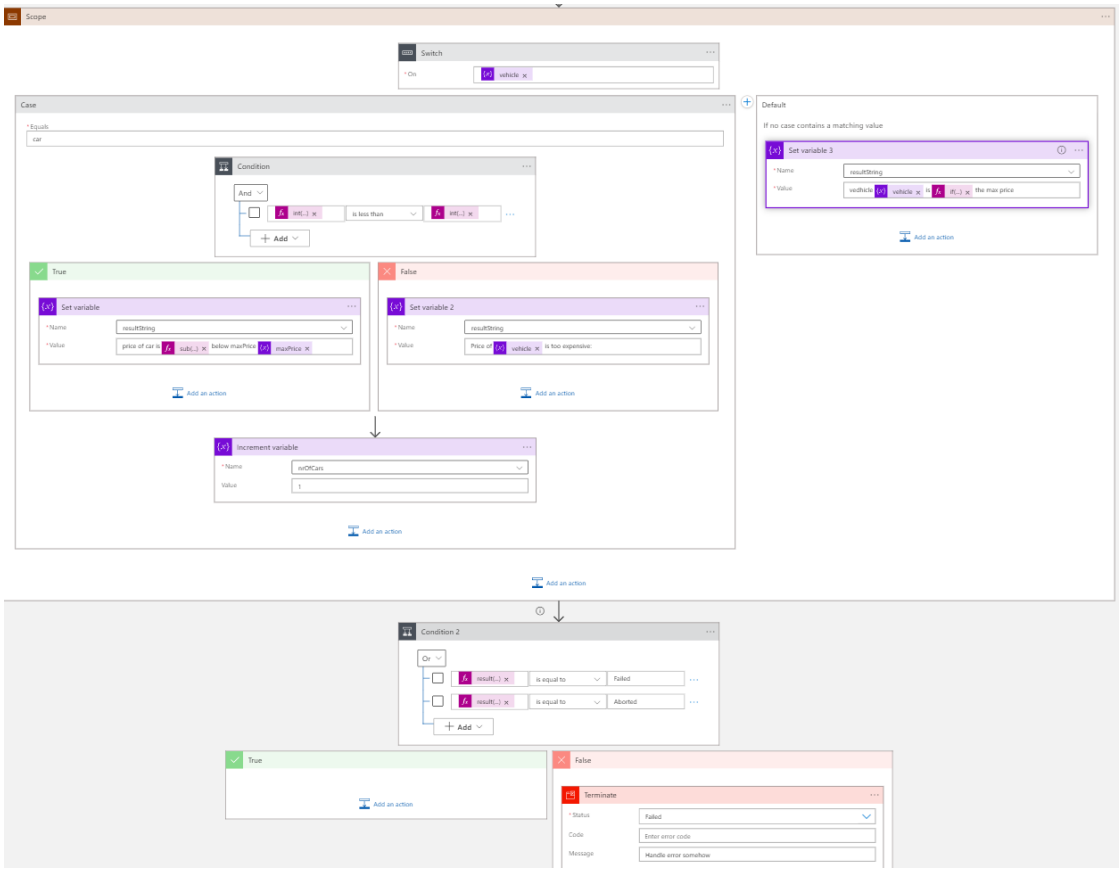

Figure 7.1: The price check operation created using Logic Apps alone. Note the number of expressions (pink blocks) and large workflow size that makes it impossible to make all actions visible in one screenshot.

Further problems arise when logic apps actions and connectors are placed inside loops, scopes, and conditions. This nested method of loops within loops makes it laborious to obtain a clear workflow overview. Viewing a scope opens up a large box in the UI that often hides or shifts the remaining operations out of view. It was not possible to acquire any clear screenshots of the entire process for this reason. Due to limitations within logic apps, some parts of the original solution require rethinking to work inside Azure. The JavaScript code operation used for regex operations is a case in point, displayed in Figure 7.2. The JavaScript code receives values from XML files based on the loop index, which is not stored as a dynamic loop value in logic apps. Creating an index variable that was incremented in each loop was revealed to be impracticable because JavaScript code actions in Azure do not support variables as of now. The problem was solved by creating an object of the index variable using a “compose” action. The result of the action, i.e., the index, could then be accessed in the JavaScript code. The described solution resolved the issue, but it is doubtful that Azure’s developers intended this way of handling data.

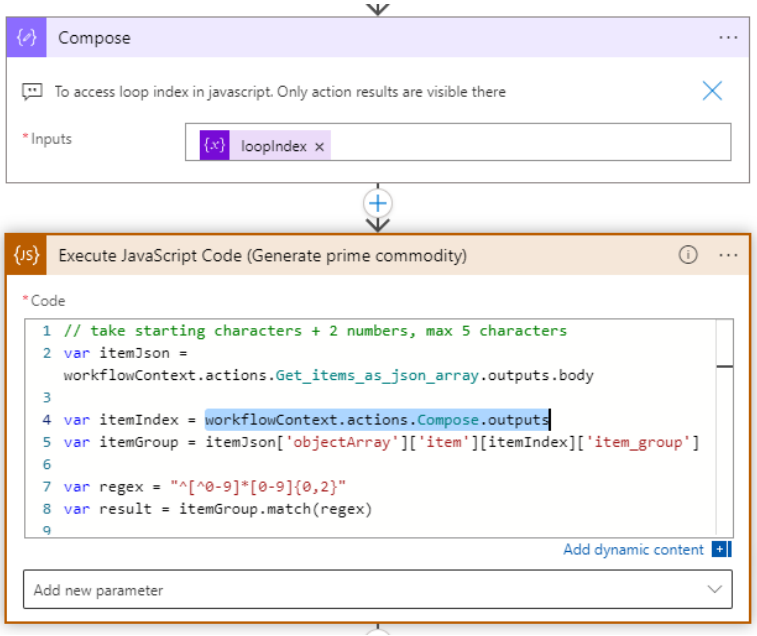

Figure 7.2: Workaround necessary to access loop index inside of JavaScript Code action in Logic Apps

The testing capabilities of logic apps are limited compared to Functions. Workflow operations can be tested individually in a workflow without executing the entire process, but most actions and connectors currently do not support this feature. This hindrance makes testing more time-consuming, since the trigger must be activated for each attempt, which means that XML files had to be moved to an input folder every time the process was to be activated. As the workflow became more elaborate, the easiest method was to create a separate Logic App for testing purposes, where only the operation in question was added. Quick task testing inside a workflow is a standard feature in other iPaaS products, such as Frends by HiQ [40]. It is safe to assume that Microsoft will improve this component of Logic Apps in the future.

7.1.3 Unresolved Constraints and Goals for Next Iteration

The pilot project exhibits the potential of a migration to a cloud platform utilizing a combination of Logic and Function Apps. In its current form, mapping data inside the Logic App workflow makes the process overly complex and challenging to manage. For a full-scale migration, adding all the requests to mapping tables in addition to the handling of purchase- and sales parts would increase the number of workflow operations to hundreds. At that stage, managing the integration process would become unwieldy. A solution for this issue would be to handle most, if not all, mapping logic inside the Function App, as proposed in 6.6.2. A switch to Azure Functions would leave the Logic App to manage the scheduling and reading of

XML files. As the legacy application is written in C#, much of the code could be reused in the functions, with the main modification involving the change from PL/SQL package procedures to API calls. Some additional data might also need to be handled, as, from my experience, the constraints regarding required fields are slightly different for SQL queries and API calls. Functions and Logic Apps are very complementary, and the difficult part is to determine which one is better suited for the use case. I am confident that a more considerable emphasis on Azure Functions would provide a better solution for the next iteration.

7.2 Cloudstep

There exists no metric for assessing the performance of a cloud migration decision process; neither can it be compared to any earlier methods used. Not many decision processes are available publicly on the web, as many are tied to companies' consulting services. Therefore, the analysis of the Cloudstep process will be highly subjective. As with the pilot project, the evaluation will be split into positives and negatives.

7.2.1 Positives

Even though application profiles are created at some level when planning a migration, at least subconsciously, systematically specifying them allows for a more precise evaluation to be made. Companies could disregard the organization profile unwittingly if they are too enchanted by the new cloud service. Identifying constraints at this early stage can eventually save time and resources. Identifying both usage and technical characteristics for the application profile makes for a well-defined description of the legacy application. This furthermore helps create the pilot project, as it is clear what functionality must be recreated in the cloud. Cloudstep's focus on iterations and constant constraints evaluations provide realistic expectations for companies looking to migrate their applications. It is unreasonable to assume that all relevant points will be detected, and the correct approach will be chosen in the first attempt. Consequently, continuous iterations are unavoidable. Compared to other migration analysis works described by Beserra et al. [15], the Cloudstep process does not require the assignment of grades or weights for decision factors. I find this advantageous as the weights can oversimplify the constraints. Using profiles and evaluating them provides a more dynamic solution.

7.2.2 Negatives

Creating the various profiles can be very laborious, especially for the cloud providers, which often encompass several services and possible options, as was the case with Azure. According to the Cloudstep model, the cloud provider should be changed if significant provider constraints exist after evaluating technical and financial constraints. In our situation, the required work would be excessive and, most importantly, redundant, as it is specifically Azure that Aveso is investigating. Modifying the step so that changes within the cloud provider are analyzed instead was an acceptable compromise. However, the most critical problem encountered with Cloudstep was the difficulty of creating an appropriate cloud provider profile. Many aspects and constraints of Azure became apparent only during pilot projects, as their finer details were hidden in various documentations that were not observed during this early phase. As such, the first iterations are destined to fail due to insufficient information. A great example of this problem is the lack of support for self-signed signatures for HTTP requests in Logic Apps. This issue only became evident when creating one of the many test projects and ultimately required redesigning the entire solution to incorporate a proxy server. Ultimately, the infinite possibilities of cloud services such as Azure and AWS combined with their multitude of documentation might not be a good fit for Cloudstep. For now, a significant reason behind the need for many iterations is the constant discovery of new constraints and alternate solutions.

7.3 Discussion

I was satisfied with the project's results, with the following possible iteration of the integration service showing great potential. The need for several iterations was not surprising, yet it was frustrating because some of the issues were due to the lack of clear documentation on Azure's side. The problem concerning self-signed certificates was not known during the planning phase of the first pilot project, but devising the solution for the constraint ended up taking up a sizable portion of the entire migration's timetable. Even so, the spontaneous decision to use Function Apps to serve as a proxy server proved to be a happy accident, as it led to further analysis of the service and ultimately the choice of switching to a primarily Azure Functions-based solution. I find that Microsoft's marketing for Azure can be unclear, as their intended use cases for Logic- and Function Apps are never disclosed. At a surface level, they may appear very similar. Based on my experiences on this project, it seems like Logic Apps are better suited for smaller-

scale integrations that can use the connectors to Microsoft's products, such as Outlook or BizTalk. Additionally, it can be challenging to decide on the optimal distribution of the services when using both. In our case, the project started with a solution based exclusively on Logic Apps, but it now seems like the best approach is to primarily use Functions, with some logic app elements for scheduling and file access.

The Cloudstep decision process proved to be a valuable resource for migration. I was not thoroughly familiar with the AvIF service, but I got a good grasp of the legacy software by creating the application profile. The small-scale pilot projects are an excellent way to test the integration, but the model would benefit significantly if more emphasis were placed on them. The best method would be to start the project by creating short test processes on the cloud provider, providing valuable insight for the provider profile. Pre-testing might be something that Cloudstep's authors [15] assume to be obvious, but it should nevertheless be pointed out. Cloudstep displays various shortcomings when applying it to a large and modern cloud service like Azure. The decision process is intended to be applied in general contexts and domains and is relatively old, released in 2012. Therefore, it can be unreasonable to expect a fit-all migration framework. It is unsurprising that companies offer consulting and migration services for cloud migrations to complex environments for which straightforward models such as Cloudstep are ill-suited.

8. Conclusion

This thesis has examined the cloud migration of a legacy integration service to Azure using the Cloudstep decision process. The main reasons for migrating to the cloud were increased flexibility and easier access. An ETL batch integration application by Aveso was selected as the legacy program to be investigated. The AvIF application is hosted on a remote server and parses XML files in a folder, transforming the data and sending them to an IFS ERP. When moving to Logic Apps, Microsoft's iPaaS, changes were necessary. The code from the legacy application had to be remade in Logic Apps' visual workflow system, and instead of PL/SQL operations, API calls were utilized. Profiles for the organization, legacy application, and cloud provider were created as described in the Cloudstep paper [15]. A pilot project was executed to examine the cloud platform's suitability further.

A pilot project was completed, where approximately half of the legacy application was replicated successfully with a combination of Logic- and Function Apps. After several iterations, it became apparent that the data operations necessary for the integration become cumbersome to implement via the visually controlled Logic Apps Designer. If the entire legacy application were to be migrated to Logic Apps, the number of operations would make the process laborious to manage. Therefore, a more significant emphasis on Function Apps would benefit the solution as it supports C# and would thus allow for reusing much of the legacy application's code. As it now stands, the Function App-based solution is the objective for the next iteration, which by all accounts should be the final rendition of the integration service.

The shift from Logic Apps to Function Apps during the migration process displays the obstacles faced when planning and executing a migration to cloud services. The services can seem suited for similar tasks at face value, but their strengths and weaknesses become evident once implementations start. The same fact applies to discovering limitations of actions and connectors in Logic Apps, causing increased complexity and workarounds that gradually increase execution costs. The monthly cost of the pilot project was approximately 136€. A full-scale migration would raise the price by 10-15€, with the non-obligatory proxy solution taking up most expenses. A shift to a Function App-based solution would alter the monthly costs slightly, but it would still be in the same price range. Overall, the monthly cost of below 150€ is not unreasonable compared to that of rivaling cloud providers.

The Cloudstep decision process was beneficial, albeit not as suitable for migrations to platforms allowing numerous potential implementation methods. By developing the legacy application and cloud provider profiles, constraints could be detected and evaluated. The most glaring constraint concerned communication and the lack of support for self-signed certificates for HTTP requests. The issue was solved by creating a proxy using Azure Functions. There were no constraints in critical areas such as security and availability since Azure's security features and SLAs were deemed satisfactory. Creating an accurate cloud provider profile is impossible without first creating small-scale tests to understand the various available features better. Some steps can seem excessive, such as changing the cloud service in case of provider constraints. In these cases, it would be more suitable to first change the solution within the cloud platform to avoid the time-consuming task of creating a new profile for an entirely different provider.

8.1 Further Research

Decision models tailored for the cloud provider should be developed for optimal migration analysis. An all-purpose model such as Cloudstep was not designed for services like Azure, where implementation methods are limitless. As the trend of cloud computing continues to grow, progressively more businesses migrate their applications to platforms such as AWS and Google Cloud in addition to Azure. It would be advantageous for customers if providers offered precise decision models and example scenarios to their potential clients without any need for joining programs or paying for consultation services. Migration models explicitly created for Azure undoubtedly exist, as is evident by many associated consulting services available online. A common factor for all these models and tutorials is that they start from the assumption that the platform's suitability has already been established. Tailored versions of Cloudstep could be explicitly created for each cloud service, although it is ultimately up to providers to create official decision models for their services. Offering more precise comparisons between services, even on the same platform, would be a step in the right direction.

9. Molnmigrering till Azure Logic Apps: en fallstudie med Cloudstep-beslutsprocessen

9.1 Introduktion

Användningen av molntjänster har ökat alltmer under det senaste årtiondet. Det är lättare än någonsin att närsomhelst få åtkomst till högpresterande hårdvara. Molntjänsters många fördelar, såsom lägre förskottsavgifter och högre nivå av skalbarhet, har lett till att många företag *migrerar* sina program till molnet. En migrering till molnet är i många fall inte oproblematisk. Med avseende på det stora antalet olika molntjänster kan det vara svårt att hitta den optimala lösningen till företagets behov. I allmänhet finns det få modeller för att avgöra en molntjänsts lämplighet för ett existerande program, och även om olika bolag erbjuder djupgående handledning i de olika stegen av en migrering, utgår de från att kunden redan bekräftat att tjänsten är rätt för dem.

Aveso är ett it-företag med fokus på *systemintegration* och konsultation. Bolaget undersöker kontinuerligt olika sätt att modernisera sina program, och ett av deras program har valts som huvudkandidat till migrering till molnet. I denna avhandling analyseras hur Avesos systemintegrationsprogram skulle fungera på Azure, Microsofts molnplattform. Dessutom kommer en generisk beslutsprocess, Cloudstep, att användas för att noggrannare granska användbarheten av en sådan process i dessa specifika fall. Brister och styrkor med Cloudstep kommer att lyftas fram, samt förbättringsförslag som kan hjälpa företag som tänker utföra liknande molnmigreringar i framtiden.

9.2 Molntjänster

Att köra program i egna lokaler på bestämd hårdvara, det vill säga *on-premises*, har länge varit normen för företag. Det är relativt nyligen som *datormoln* har blivit ett tänkbart alternativ. Teknologin introducerades för allmänheten av Amazon år 2002 i form av Amazon Web Services (AWS), och andra storbolag har senare introducerat sina egna molnbaserade plattformar. Organisationen *National Institute of Standards and Technology* (NIST) definierar molntjänster som ”en modell för att möjliggöra bekväm nätverksåtkomst till en delad pool av konfigurerbara datorresurser på begäran varifrån som helst. Dessa datorresurser, som är i form av nätverk, servrar, lagring, applikationer och tjänster, kan snabbt provisionernas och

släppas med minimal hantering eller interaktion från tjänsteleverantören (övers.)" [2]. NIST beskriver molntjänster noggrannare med hjälp av vad som ofta kallas för 5-4-3-principerna för moln. De listar fem egenskaper som är väsentliga för alla molntjänster, varav den första är självbetjäning på begäran, det vill säga datorkapacitet ska kunna levereras utan kontrollåtgärder från människor. Omfattande tillgänglighet via nätverk och resursfördelning mellan användare är de två följande aspekterna. De två sista egenskaperna berör snabb elasticitet och mätning av resursanvändning. NIST tar även upp fyra implementeringsmodeller. I ena änden av spektrumet finns *privata moln*, ägnade för internt bruk i företag, och på andra änden finns *offentliga moln* som är öppna för allmänt bruk. Mellan dessa finns även så kallade *community-moln* för flera företags bruk, samt hybridlösningar. Slutligen räknar NIST upp tre servicemodeller för molntjänster: *Program som nättjänst* (*SaaS*), *platform som nättjänst* (*PaaS*) och *Infrastruktur som nättjänst* (*IaaS*). Termerna beskriver vilken nivå av abstraktion som tjänsten har. Med SaaS använder slutanvändaren ett färdigt program som körs på programuthyrarens servrar, medan PaaS låter kunden hyra servrar för att köra sina egna applikationer.

Molntjänster har flera fördelar jämfört med körning lokalt. Resurser på molnet är tillgängliga närsomhelst varifrån som helst i världen. Kapitalkostnaderna är lägre eftersom ingen fysisk hårdvara behöver införskaffas och resurserna kan snabbare läggas till eller tas bort enligt behov. Företag behöver inte heller ha samma nivå av sakkunskap för att komma i gång med serverinstallationer. De två mest framträdande orsakerna till oro hos användare som flyttar sina program till molnet är säkerheten och integriteten av deras data [8]. Säkerheten som molntjänster erbjuder kan dock i vissa fall överskrida den av ett företags lokala hårdvara, och leverantörerna av molntjänster erbjuder sina specifika *tjänstenivåavtal* (SLA).

9.3 Migrering till Molnet

För att hjälpa kunder med att migrera sina program till molnet har tjänsteleverantörer såsom Amazon och Microsoft skapat sina egna migreringsmodeller [13] [14]. Fastän modellerna till en viss grad är skräddarsydda för specifika molntjänster, innehåller de oftast ändå följande steg: utvärdering, planering, utförande, validering och underhåll. I utvärderingsfasen utförs affärs- och teknikanalyser samt en uppskattning av arbetsbördan. Vid planeringen väljs en migrationsstrategi, vilka enligt e-boken "*Migrating to AWS: Best Practices and Strategies*" av Stephen Orban kan delas in i sex kategorier [14]. Med "*lyft och*

flytta”-strategin migreras IT-systemet utan några stora ändringar, medan *refaktorisering* innebär omstrukturering av hela programmet så att det bättre ska passa molnet. En blandning mellan dessa två är ”*lyft, knåpa och flytta*”. Kunden kan även välja att köpa en färdig tjänst som motsvarar hens existerande program. De två sista strategierna berör fall där ingen migration utförs eftersom programmet antingen är för gammalt, eller så fungerar det bättre då det körs lokalt. Då strategin valts utförs själva migreringen. Därefter kan validering och optimering göras kontinuerligt för att förbättra resultaten. Då programmet körs på molnet kräver det slutligen någon form av styrning och övervakning.

Molnmigreringsmodellen Cloudstep introducerades av Patricia V. Beserra et al. år 2012 [15]. Cloudstep är ägnad för att hjälpa applikationsutvecklare och projektledare med att fatta lämpliga beslut angående moln. Med hjälp av att skapa olika profiler kan de viktigaste aspekterna av organisationen, applikationen och molntjänsten identifieras. Genom att iterativt följa Cloudstep-modellen ska användare bättre få en bild av om molntjänsten passar deras applikationer.

9.4 Systemintegration

I och med att företag växer producerar de även mer data. Informationen sparas ofta lokalt eller på moln, i en kombination av olika format. Det har blivit fördelaktigt för program och företag att dela på denna information. Data från olika källor är i många fall oförenliga med varandra, och måste i sådana fall behandlas på något sätt innan de kan kombineras. Detta är kärnan av systemintegration. I boken ”*Principles of Data Integration*” definierar AnHai Doan et al. systemintegration noggrannare genom dess mål, att ”erbjuda enhetlig tillgång till en uppsättning autonoma och heterogena datakällor (övers.)” [18]. Autonomi avser att de administrativa rättigheterna till källorna kan variera, medan heterogenitet syftar på att datakällorna skiljer sig från varandra. En vanlig form av systemintegration är ETL, *hämtning, konvertering och lagring* (eng. *extract, transform, load*). Vid hämtning verifieras informationen, som sedan exporteras till ett s.k. uppställningsområde där den konverteras enligt vissa formler. Då informationen lagras i slutsystemet kan ytterligare validitetsgranskningar ännu utföras. Man brukar skilja mellan *satsvis bearbetning* och *realtidsintegration*. Satsvisa bearbetningar är asynkrona och överför data enligt en förutbestämd tidtabell, där termen sats betecknar den stora datamängden som bearbetas. Löpande bearbetning används i fall där gränssnitt väntar på att transaktionen ska slutföras i alla inblandade system innan den avslutas, exempelvis vid kreditkortsbetalningar.

Ett populärt sätt att få tillgång till uppgifter från olika tjänster är genom ett *API*, ett *applikationsprogrammeringsgränssnitt* som tjänsteleverantören skapat. Genom API:er kan företag på ett reglerat sätt tillhandahålla data och tjänster åt andra användare, utan att de behöver förstå den underliggande strukturen. Begäran till webb-API:er görs oftast med hjälp av *HTTP*-förfrågningar. Det vanligaste mönstret för att skapa API:er är *REST* (*Representational State Transfer*), som introducerades av Roy Felding år 2000 [28]. Data från ett API skickas ofta tillbaka i form av *JSON* (*Javascript Object Notation*) eller *XML* (*Extensible Markup Language*).

9.5 Utförande och Implementation

De första stegen i Cloudstep-beslutsprocessen är att definiera ***organisationens profil*** samt granska om det finns några begränsningar i migreringen. Orsaken till att Aveso undersöker en migrering till molnet är för att modernisera dess applikationer och således locka nya kunder. Avesos personal är bekanta med Azure-plattformen, och företaget begränsas inte av några lagar om den fysiska platsen av datalagring.

Som följande skapas en ***profil av applikationen***, Aveso Integration Framework (***AvIF***). AvIF-gränssnittet läser XML-filer satsvis från en förutbestämd mapp. Informationen behandlas och skickas därefter till ett ERP-system som befinner sig på samma värddator. AvIF-tjänster kontrolleras med hjälp av ett separat gränssnitt, AvIF Manager, och noggranna inställningar såsom adresser för indata görs i en skild konfigurationsfil. Eftersom AvIF-applikationer installeras på kunders servrar, kommer administratörer endast åt dom med hjälp av fjärrskrivbord. AvIF-applikationer kodade i C#, och de kontaktar ERP-system med hjälp av PL/SQL-operationer.

Vid skapandet av ***profilen för molntjänsteleverantören*** granskas Logic Apps, Azures integrationsorienterade PaaS. Tjänster i Logic Apps skapas med hjälp av visuella *arbetsflöden* bestående av en trigger, följt av en serie steg innehållande processer eller händelser. Dessa händelser kan vara antingen inbyggda, till exempel beräkningar och hantering av variabler, eller så kan de vara *connectors* som är anslutna till externa tjänster, såsom Outlook. För projektet utnyttjas en trigger som är ansluten till en katalog och startar programmet då en ny fil upptäcks. Hantering av data sker nästan helt via inbyggda funktioner, och information skickas till ERP-systemet via REST API-förfrågningar. Kostnaden för Logic Apps påverkas av vilken prisplan som väljs. I vår fallstudie betalas tjänsten enligt användning, vilket estimeras blir 10€/månad. Azures SLA lovar en tillgänglighet på 99,9 %, samt erbjuder Microsoft flera säkerhetstjänster som kan användas utöver de inbyggda.

Då ***profilerna för applikationen och molntjänsteleverantören jämförs*** är det tydligt att den största begränsningen berör kommunikation: HTTP-förfrågningar i Logic Apps stöder inte självsignerade certifikat, som används av Avesos testserver. För att kringgå detta används en proxyserver, denna gång skapad med Azure Functions. Användningen av en mellanserver kan leda till ekonomiska begränsningar, men i detta fall fortsätter vi processen. Ifall det fortfarande finns begränsningar, ska ***andra molntjänster undersökas*** enligt Cloudstep. I vår fallstudie är detta redundant, eftersom det specifikt är Azure som undersöks. Därför modifierades steget en aning, och endast alternativa lösningar inuti Azure undersöks. En migreringsstrategi väljs också, vilket i Avesos fall är återköp.

I nästa steg ska ett småskaligt ***pilotprojekt*** utföras för att kontrollera att molntjänsten fungerar som förväntat. Cirka hälften av AvIF-applikationen återskapades med Azure Logic Apps och Functions, och alla dess väsentliga funktioner överfördes framgångsrikt. Det slutliga arbetsflödet innehöll 37 händelser, och dess månadskostnad estimerades till 136€ (10€ utan proxyserver).

9.6 Analys och diskussion

Efter pilotprojektet kan nästa iteration av projektet planeras. Utgående från processen som skapades i Logic Apps är det uppenbart att hanteringen av information fungerar bättre inuti Azure Functions. Därför är planen för nästa iteration att övergå till en mera Functions-baserad lösning. Den tydligaste positiva aspekten med Logic Apps var dess visuella verktyg för att skapa arbetsflöden. Jämfört med utveckling av applikationer i exempelvis C# så behövs knappt någon kodkunskap alls med Azure, och Microsoft erbjuder flera mallar för att komma i gång. Dock krävs det många steg för att utföra till synes enkla beräkningar, vilket gör att traditionell kod är mer koncis och lättläslig för större projekt. Cloudstep stödde analysprocessen genom framställningen av profiler som kunde jämföras för att direkt upptäcka begränsningar. Dessutom var det lättare att skapa ett testprojekt då applikationens väsentliga funktioner var etablerade. Dessvärre uppvisar Cloudstep svagheter då modellen appliceras på en modern molntjänst. Skapandet av profiler är tidskrävande, och flera begränsningar blev uppenbara först efter flera iterationer. Detta beror delvis på en brist av tydlig dokumentation av Azures funktioner, samt avsaknaden av testande i ett tidigt skede av Cloudstep-processen.

9.7 Avslutning

I denna avhandling har en migrering till molnet undersökts med hjälp av Cloudstep-beslutsprocessen. Lämpligheten av Cloudstep analyserades även då det tillämpas på en modern molntjänst såsom Microsoft Azure. Beslutsprocessen följdes, varefter ett småskaligt projekt utvecklades till Azure Logic Apps. Efter en utvärdering av testprojektet blev det tydligt att ett större fokus på Azure Functions skulle gynna projektet, eftersom C#-koden från den ursprungliga applikationen då delvis kunde återanvändas. Svåra datahanteringsoperationer kräver flera steg i Logic Apps-arbetsflöden och gör hela processen svårläslig.

Profilerna som skapades för Cloudstep hjälpte migreringen, men beslutsprocessen kunde uppdateras för att bättre passa nya molntjänster. Små testprojekt borde skapas direkt för att upptäcka begränsningar och undersökandet av andra molntjänster kunde ersättas med alternativa lösningar inom samma plattform. I framtiden kunde skräddarsydda Cloudstep-beslutsprocesser skapas för specifika molntjänster, men tjänsteleverantörer kunde även förbättra sina egna analysprocesser.

References

[1] Gartner, *Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 23% in 2021*, Gartner.com, 21-Apr-2021. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-in-2021 [Accessed: 25-May-2021]

[2] T. Grance and P. Mell, *The NIST Definition of Cloud Computing*, Computer Security Division, NIST, Gaithersburg, Sep-2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf [Accessed 25-May-2021]

[3] S. Vennam, *Cloud Computing*, IBM.com, 18-Aug-2020. [Online]. Available: https://www.ibm.com/cloud/learn/cloud-computing [Accessed 25-May-2021]

[4] K. Chandrasekaran, 2015. *Essentials of cloud computing* (1st edition). Boca Raton: CRC Press. https://abo.finna.fi/Record/abo_electronic_aa.9913542526305972

[5] Microsoft, *What is a private cloud?* [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-a-private-cloud/ [Accessed 26-May-2021]

[6] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra and B. Hu, *Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends*, 2015 IEEE 8th International Conference on Cloud Computing, 2015, pp. 621-628 https://ieeexplore.ieee.org/document/7214098

[7] Gartner, *Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 18% in 2021*, Gartner.com, 17-Nov-2020. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-18-percent-in-2021 [Accessed 27-May-2021]

[8] S. Murugesan & I. Borislava, *Encyclopedia of cloud computing*. 2016, Chichester, West Sussex, United Kingdom; Hoboken, NJ: Wiley. https://abo.finna.fi/Record/abo_electronic_aa.9913459025005972

[9] Microsoft, *Azure.* [Online]. Available: https://azure.microsoft.com/ [Accessed 3-Dec-2021]

[10] Janakiram M.S.V., *A Look Back At Ten Years Of Microsoft Azure*, Forbes, 3-Feb-2020. [Online]. Available: https://www.forbes.com/sites/janakirammsv/2020/02/03/a-look-back-at-ten-years-of-microsoft-azure/?sh=7a1363254929 [Accessed 27-May-2021]

[11] Talend Knowledge Center, *What is a Legacy System?* [Online] Available: https://www.talend.com/resources/what-is-legacy-system/ [Accessed 27-May-2021]

[12] A. Dapre, *Microsoft Azure at Sibos 2018 – Intelligent Banking*, Microsoft Industry Blogs. [Online]. Available: https://cloudblogs.microsoft.com/industry-blog/financial-services/2018/10/16/microsoft-azure-at-sibos-2018-intelligent-banking [Accessed 27-May-2021]

[13] Microsoft, *Cloud Migration Simplified*, 18-May-2020. [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-migration-simplified/ [Accessed 27-May-2021]

[14] S. Orban (based on blog series by), *Migrating to AWS: Best Practices and Strategies*, Amazon Web Services. [Online] https://d1.awsstatic.com/Migration/migrating-to-aws-ebook.pdf

[15] P. V. Beserra, A. Camara, R. Ximenes, A. B. Albuquerque and N. C. Mendonça, *Cloudstep: A step-by-step decision process to support legacy application migration to the cloud*, 2012 IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012, pp. 7-16 https://ieeexplore.ieee.org/document/6392602

[16] A. S. Roy, *How does facebook handle the 4+ petabyte of data generated per day? Cambridge Analytica - facebook data scandal*, Medium.com, 16-Sep-2020, [Online]. Available: https://medium.com/@srank2000/how-facebook-handles-the-4-petabyte-of-data-generated-per-day-ab86877956f4 [Accessed 27-May-2021]

[17] Oracle, *What is Big Data?* [Online]. Available: https://www.oracle.com/big-data/what-is-big-data/ [Accessed 27-May-2021]

[18] A. Doan, A. Halevy, & Z. G. Ives, 2012. *Principles of data integration*. (1st edition). Waltham, Mass.: Morgan Kaufmann. https://abo.finna.fi/Record/abo_electronic_aa.9913437576105972

[19] Latt, *An overview of Data Lake concepts and architecture on AWS and Azure*, FAUN, 13-Sep-2020. [Online]. Available: https://faun.pub/an-overview-of-data-lake-concepts-and-architectures-on-aws-and-azure-f485ed5110e2 [Accessed 27-May-2021]

[20] J. Kutay, *ETL vs ELT: Key Differences and Latest Trends.* Striim Blog, 5-Mar-2021. [Online]. Available: https://www.striim.com/etl-vs-elt-2/ [Accessed 3-Dec-2021]

[21] R. Sherman & C. Imhoff, 2015. *Business intelligence guidebook: From data integration to analytics*. 1st edition. Waltham, Massachusetts: Morgan Kaufmann. https://abo.finna.fi/Record/abo_electronic_aa.9913431336005972

[22] A. Reeve, 2013. *Managing data in motion: Data integration best practice techniques and technologies*. 1st edition. Waltham, Mass.: Morgan Kaufmann. https://abo.finna.fi/Record/abo_electronic_aa.9913435685705972

[23] R. F. v. d. Lans, 2012. *Data virtualization for business intelligence architectures: Revolutionizing data integration for data warehouses*. 1st edition. Amsterdam; Boston: Elsevier/MK. https://abo.finna.fi/Record/abo_electronic_aa.9913439498705972

[24] M. Reddy, 2011. *API design for C++*. 1st edition. Boston: Elsevier/Morgan Kaufmann. https://abo.finna.fi/Record/abo_electronic_aa.9913448889905972

[25] D. Jacobson, G. Brail and D. Woods. *APIs: A Strategy Guide*, Dec-2011. Sebastopol, CA: O'Reilly. https://books.google.fi/books?id=pLN7BxMTg7IC

[26] M. Medjaoui, E. Wilde, R. Mitra, M. Amundsen, *Continuous API Management: Making the Right Decisions in an Evolving Landscape*, 14-Nov-2018. Sebastopol. CA. O'Reilly. https://books.google.fi/books?id=P9B5DwAAQBAJ

[27] A. Macoveiciuc, *Beginner's Guide to APIs, Protocols and Formats*. 29-Apr-2020. [Online]. Available: https://frontend-digest.com/beginners-guide-to-apis-protocols-and-data-formats-f80cf7f30425 [Accessed 3-Dec-2021]

[28] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Dissertation, Information and Computer Science. University of California, Irvine, 2000 https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[29] *What is REST*, Restfulapi.net, Updated 19-Oct-2021. [Online]. Available: https://restfulapi.net/ [Accessed 3-Dec-2021]

[30] Swaggerhub, *swagger.io.* [Online]. Available: https://swagger.io/tools/swaggerhub/ [Accessed 3-Dec-2021]

[31] *JSON vs XML,* Restfulapi.net, Updated 27-Sep-2021. [Online]. Available: https://restfulapi.net/json-vs-xml/ [Accessed 3-Dec-2021]

[32] B. Cooksey, *An Introduction to APIs, Chapter 3: Data Formats.* Zapier, 22-Apr-2014. [Online]. Available: https://zapier.com/learn/apis/chapter-3-data-formats/ [Accessed 3-Dec-2021]

[33] *About IFS*, Ifs.com. [Online]. Available: https://www.ifs.com/company/about-ifs/ [Accessed 3-Dec-2021]

[34] J. Fancey, *New Logic Apps runtime, performance and developer improvements.* Microsoft Tech Community, 21-Sep-2020. [Online]. Available: https://techcommunity.microsoft.com/t5/azure-developer-community-blog/new-logic-apps-runtime-performance-and-developer-improvements/ba-p/1645335 [Accessed 3-Dec-2021]

[35] Microsoft Docs, *What is Azure Logic Apps?,* Updated 26-Oct.2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview [Accessed 3-Dec-2021]

[36] Microsoft, *Azure Logic Apps*, Azure. [Online]. Available: https://azure.microsoft.com/en-us/services/logic-apps/#overview [Accessed 3-Dec-2021]

[37] Microsoft, *Logic Apps pricing*, Azure. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/logic-apps/ [Accessed 3-Dec-2021]

[38] Microsoft, *Azure functions pricing*, Azure. [Online]. Available: https://azure.microsoft.com/en-gb/pricing/details/functions [Accessed 3-Dec-2021]

[39] Microsoft Docs, *Azure Infrastructure Security*, Microsoft Documentation, 11-Nov-2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/security/fundamentals/infrastructure [Accessed 3-Dec-2021]

[40] HiQ, *Frends* [Online]. Available: https://frends.com/ [Accessed 3-Dec-2021]

Appendix

Appendix A: OpenAPI specification for PATCH-call in IFS ERP. The specification contains API calls with example responses. Attributes and formats of the response can be seen in the 'Models' box (bottom right). OpenAPI file is displayed using **SwaggerHub** [30]

Appendix B: Mapping table for AvIF integration (reduced to only include relevant information for pilot project).

XML file field	Constraints	Modifications	IFS field
<item_code>	Mandatory field.		**Part no**
<item_ver> **<item_create_date>**	Create_date Mandatory field	Form revison text using following logic: Rev. <item_ver> (<item_create_date>) Where *item_ver* default is 1 and *item_create_date* is date where time has been removed	Part Revision **Revision_text**
<item_type>	Mandatory field	Determines what headers are created in IFS. Uses mapping table	Not brought into IFS
<item_group>	Mandatory field. Header group (5 first characters) must exist in IFS.	From 2 part groups. Part group 1: • First 5 chars • All letters from start • After that, max 2 numbers • Example: AB12 (From AB1234) Part group 2: • 5 first characters	Inventory Part **Prime_commodity** (header group 1) **Second_commodity** (header group 2)
<item_status>	Must be *ACCEPTED.*		
<item_magnitude>	Mandatory field .	Convert all characters to lowercase. Change: • Mm => m • Kpl => pcs	Part Catalog **Unit_code** Inventory Part **Unit_meas**
<item_desc1><item_desc2>	Item_desc1 Mandatory field .	Desc1 + desc2 separated with whitespace	Part Catalog / Inventory Part / Purchase Part **description**
<mass>			Part catalog **Weight_net**
<drawing_number>			Inventory part **Type_designation**
<length> **<width>**		Form combined value: L=<length> x W=<width>	Inventory Part **Dim_quality**

Appendix C: Full code for part catalog PATCH API call using Azure Function App

```
public static class PartCatalogPatch
    {
        [FunctionName("PartCatalogPatch")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "patch", Route = null)] HttpRequest req,
            ILogger log)
        {
            ApiClass.InitializeClient();

            var reader = new StreamReader(req.Body);
            reader.BaseStream.Seek(0, SeekOrigin.Begin);
            var rawJson = reader.ReadToEnd();

            PartCatalog part = JsonConvert.DeserializeObject<PartCatalog>(rawJson);

            bool success = ApiCalls.UpdatePartCatalog(part.PartNo, rawJson, log).Result;

            return new OkObjectResult(success ? "Success" : "Failed");
        }
    }

public static void InitializeClient()
    {
        //unsafe!
        var handler = new HttpClientHandler()
        {
            ServerCertificateCustomValidationCallback = HttpClientHandler.DangerousAcceptAnyServerCertificateValidator
        };

        ApiClient = new HttpClient(handler);
        ApiClient.DefaultRequestHeaders.Accept.Clear();
        ApiClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
        ApiClient.BaseAddress = new Uri(Environment.GetEnvironmentVariable("BaseUrl", EnvironmentVariableTarget.Process));

        string basicAuth = Environment.GetEnvironmentVariable("BasicUser", EnvironmentVariableTarget.Process) + ":" +
Environment.GetEnvironmentVariable("BasicPassword", EnvironmentVariableTarget.Process);
        var byteArray = Encoding.ASCII.GetBytes(basicAuth);
        ApiClient.DefaultRequestHeaders.Authorization = new System.Net.Http.Headers.AuthenticationHeaderValue("Basic",
Convert.ToBase64String(byteArray));


    }

public static async Task<bool> UpdatePartCatalog(string partNo, string partJson, ILogger log)
    {
        string url = "PartHandling.svc/PartCatalogSet(PartNo='" + partNo + "')";
        try
        {
            ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
            var content = new StringContent(partJson.ToString(), Encoding.UTF8, "application/json");
            ApiClass.ApiClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

            using (HttpResponseMessage response = await ApiClass.ApiClient.PatchAsync(ApiClass.ApiClient.BaseAddress + url, conte
            {

                if (response.IsSuccessStatusCode)
                {
                    return true;
                }

                else
                {
                    Console.WriteLine("Error");
                    throw new Exception(response.ReasonPhrase);
                }
            }
        }

        catch (Exception e)
        {
            if (e.Source != null)
                Console.WriteLine("Exception {0} source: {1}, {2}", e.InnerException, e.Message, e.HelpLink);
            throw;
        }
    }
```