# Serverless Computing Use Cases – Benefits and Disadvantages

Jonas Kylliäinen
Master's Thesis in Computer Engineering
Supervisor: Adnan Ashraf
Faculty of Science and Engineering
Information Technologies
Åbo Akademi University
2022

# Abstract

Serverless Computing is a form of cloud service where the cloud vendor offers a platform with a simplified development and deployment environment, allowing developers to focus primarily on implementing business logic. Moreover, the operation burden is reduced as the cloud vendor manages all the inner workings of the platform. The level of abstraction has also made the consumption-based billing-model possible, in which the customer only pays for the computational resources consumed by their applications.

This thesis explores the benefits and disadvantages of implementing business solutions in the cloud using serverless platforms as opposed to more conventional application development. Three example use cases are investigated, each having a conventional implementation and serverless alternative. The serverless implementations were developed using cloud services available in Microsoft Azure, whilst the conventional ones utilised Java combined with Spring Boot.

The serverless implementations demonstrated that the platforms made it effortless to implement certain features, as the complexity was managed by the vendor. Moreover, the characteristics of the platforms made them particularly suitable for applications featuring bursty workloads. However, there are drawbacks as well, a considerable one being the hard dependencies to the platforms that the serverless implementations featured. These dependencies create an effective vendor lock-in, tying the customer to the cloud vendor.

**Keywords:** Serverless, Cloud computing, Azure, Java, Spring Boot

# Contents

# List of Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **AWS** | Amazon Web Services |
| **BIS** | Business Information System |
| **BPMN** | Business Process Model and Notation |
| **CRUD** | Create, Read, Update and Delete |
| **DB** | Database |
| **DTO** | Data Transfer Object |
| **ERP** | Enterprise Resource Planning |
| **HTTP** | Hypertext Transfer Protocol |
| **JAXB** | Jakarta XML Binding |
| **JPA** | Jakarta Persistence API |
| **JSON** | JavaScript Object Notation |
| **OS** | Operating System |
| **REST** | Representational State Transfer |
| **RPA** | Robotic Process Automation |
| **RU** | Request Unit |
| **NoSQL** | Not only SQL |
| **SQL** | Structured Query Language |
| **SFTP** | SSH File Transfer Protocol |
| **SMTP** | Simple Mail Transfer Protocol |
| **TEO** | The Example Organisation |
| **URL** | Uniform Resource Locator |
| **VAT ID** | Value Added Tax Identifier |
| **XML** | Extensible Markup Language |
| | |
| **FaaS** | Function-as-a-Service |
| **PaaS** | Platform-as-a-Service |
| **IaaS** | Infrastructure-as-a-Service |

# List of Figures

# List of Listings

# List of Tables

# 1. Introduction

*Serverless Computing* is an emerging technology which promises to simplify application development and deployment. With advancements in cloud computing and virtualisation techniques during the previous decade, cloud vendors have begun introducing various serverless platforms. In these platforms, the cloud vendors themselves manage the operational aspects of executing the applications in the cloud. The inner workings of the platforms and the underlying infrastructure have been abstracted away from the developers, allowing them to build applications using high-level abstractions instead. Therefore, developers can focus primarily on implementing business logic [1].

This, combined with the consumption-based billing model, in which the cloud vendor only bills the customers when the application is executed, makes serverless platforms a lucrative alternative to conventional applications. However, serverless technologies are still in their early stages. Although the platforms provide many benefits, there are disadvantages as well, particularly regarding certain aspects. Therefore, when to utilise serverless technologies is not always clear.

## 1.1 Motive

This thesis aims to explore the benefits and disadvantages that serverless technologies provide, as well as how these compare against more conventional application development methods. To make these comparisons, the author has implemented a serverless and a conventional implementation for three example business use cases. The use cases aim to demonstrate a set of scenarios which businesses may face.

The implementations and their development processes are then compared from different perspectives to demonstrate the advantages and disadvantages of each approach. The serverless implementations presented in this thesis are developed using Microsoft Azure's serverless platforms, while the conventional counterparts are developed using Java in combination with Spring Boot.

## 1.2 Structure

This first chapter introduced the concept of serverless computing along with the purpose of this thesis. The topic of serverless computing is further expanded in chapter 2. Moreover, the chapter presents other background information related to the thesis and a literature review of the topic. Chapter 3 gives an overview of the services in Azure which are used in the serverless implementations. The use cases investigated in this thesis are presented in chapter 4. The conventional and serverless implementations of these use cases are then presented in chapter 5. These implementations and their development processes are then compared in chapter 6. Chapter 7 discusses the benefits and disadvantages of serverless more broadly based on the findings. Finally, chapter 8 presents some concluding remarks about the work.

# 2. Background and Related Work

## 2.1 Serverless Computing

With the introduction of cloud computing, new ways of deploying and developing applications have emerged that fall under the umbrella of Serverless Computing, also simply known as serverless. In the context of cloud computing, serverless does not mean that there are no servers used to execute code, but rather that the servers and other underlying infrastructure have been abstracted away. This allows developers to deploy and maintain applications without having to provision or manage servers or other resources. The cloud vendor dynamically handles the provisioning of computing resources as needed when the application is to be executed [2].

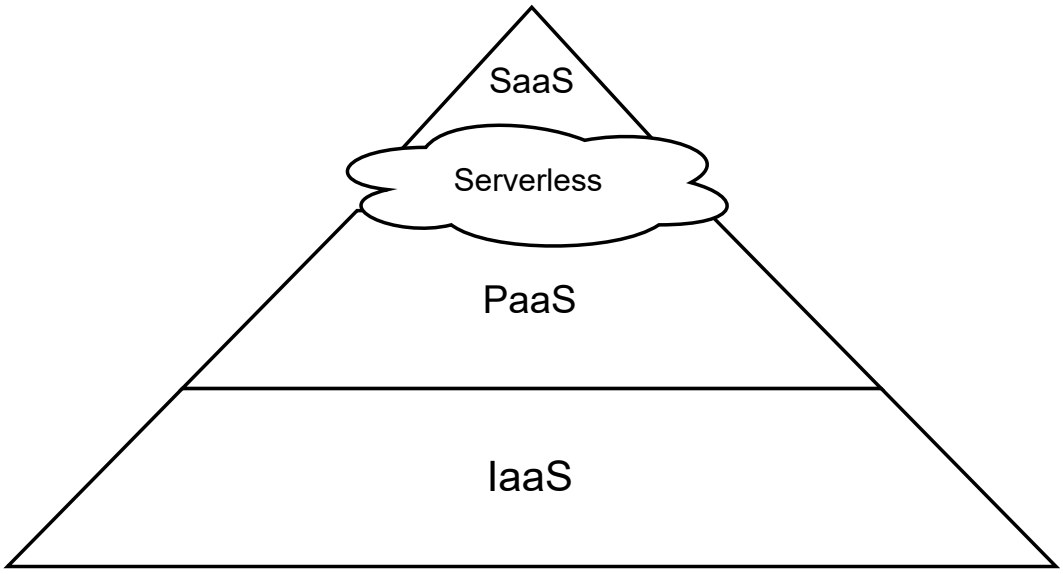


Figure 2.1: Serverless is said to sit between PaaS and SaaS, but the line is blurry

What exactly constitutes serverless can be challenging to define. According to one definition [3], serverless sits somewhere in between Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). In PaaS, the servers and other underlying infrastructure are also abstracted away, allowing applications to be deployed without the developers having to maintain the environment the application runs within [1]. However, serverless takes this a step further by having the developers focus

on implementing business logic utilising higher-level abstractions, which can execute in the cloud independently. The execution of the business logic is triggered in response to events defined by the developer. The cloud provider handles the operational aspects of executing the business logic, such as allocating the required computational resources and mapping the high-level abstractions to concrete under-the-hood implementations. This reduces the operational burden on the developers whilst also removing some control [1]. The amount of control removed varies depending on the serverless platform and how the service provider has designed it.

The reduction of control brings serverless closer to SaaS, where the user has little or no control over the inner workings of the software itself. In SaaS, the provider is in charge of developing and maintaining a piece of software, which is then offered to the end users as a service. The amount of control given to the users varies depending on the configurability of the SaaS product. Some are provided as is, whilst others may even allow the users to add custom functionality. However, this functionality is ultimately limited by the domain of the SaaS product, as the execution occurs within the application context. In serverless, these kinds of limitations do not exist [3].

The most common type of serverless model is known as Function-as-a-Service (FaaS) [2]. In FaaS, developers write pieces of code in the form of functions, which can then independently be executed in the cloud. Multiple functions may be grouped together to form a more extensive application, where each function conducts a part of a larger workflow. In many cases, FaaS platforms support multiple programming languages. This allows organisations and developers to create functions utilising one or more programming languages most suitable for their needs [2][3].

Amazon was the first major cloud vendor to offer the FaaS model with the introduction of AWS Lambda [4] in November 2014 [5]. As the first serverless platform on the market, it defined many of the development, operational, and resource aspects that can be seen on other similar platforms today [3]. Other cloud vendors quickly followed suit, with both Microsoft and Google releasing their own FaaS offerings two years later, Azure Functions [6] and Google Cloud Functions [7], respectively. Since then, cloud vendors have also introduced other serverless offerings that are not FaaS based, such as Azure Logic Apps [8] from Microsoft.

Serverless platforms provide new ways of developing cloud-native applications. For developers, one of the significant benefits of serverless platforms is that they integrate well with the other services part of the cloud vendors' ecosystem [3]. For example, the platform may include built-in functionality to authenticate users or to

retrieve and save files in the cloud vendor's own storage service. This, combined with the abstraction of the operational aspects of executing the code, makes the overall development process easier and faster, thus allowing developers to focus primarily on implementing the application's business logic [9].

Serverless offerings are also characterised by a billing model which differs from traditional cloud computing services. Unlike IaaS offerings, such as virtual machines, which are typically billed based on reserved capacity, serverless platforms are generally billed based on usage. This means that the customer only pays for the resources consumed during the execution of the code, which is measured by the cloud provider. This type of billing model is possible as the cloud provider dynamically allocates the computational resources as they are needed. The specifics of how resources are billed vary depending on the serverless platform and cloud vendor. However, at least in FaaS offerings, the billing model is commonly based on the number of function executions, the execution time, and the amount of memory consumed during the execution [10][11].

This consumption-based billing model can make serverless offerings incredibly affordable. Not only does the customer avoid the need to set up any infrastructure to run their application, but they also solely need to pay for whatever resources are consumed whenever it is being executed. However, this type of billing model is not necessarily always less expensive compared to reserved capacity. In scenarios where the resource consumption on the serverless platform is high, the operational costs may be more expensive. This may occur, for instance, if an application is constantly being executed.

Another drawback of serverless platforms is a phenomenon known as cold starts. Due to the serverless application having no reserved resources in the cloud, the cloud provider has to set up and allocate resources from zero whenever the application is executed after an idling period. This can result in a performance penalty during the early stages of the execution [3].

Furthermore, serverless platforms suffer from certain resource limitations, making them unsuitable for some applications. One major limitation is the maximum execution time, which is often limited. The limit varies depending on the serverless platform. For instance, in Azure Functions, the default timeout is after 5 minutes, but this can be extended to a maximum of 10 minutes [11]. AWS Lambda offers a slightly longer maximum timeout of 15 minutes [12]. Developers need to consider this time limit when designing serverless solutions. For example, splitting larger jobs into multiple smaller ones could allow for the serverless code to execute within the time limit [1][2].

From a service management perspective, one issue with most serverless platforms is that they result in the applications being dependent on the specific platform they are developed for. This is primarily due to cloud vendors having their own set of features, tools, and requirements for their serverless platforms. The implication is that applications designed for one platform cannot be transferred to another without refactoring. For instance, a FaaS application developed for Azure Functions cannot be deployed using AWS Lambda, even if both platforms would support the same programming language. The problem is further amplified by each serverless platform integrating well with the cloud vendors' other services, many of which are often proprietary. These services must also be replaced should the application be deployed on another cloud vendor's platform. Replacing these with alternative services may result in high costs and require substantial effort [13].

This issue is known as vendor lock-in. Once a customer has begun using the services of a particular vendor, it is difficult and expensive to stop using or transfer the services to another provider. Furthermore, cloud vendors are often incentivised to design their services with this in mind to reduce customer churn [14].

## 2.2 Literature Review of Serverless Computing

The field of serverless computing is relatively new, having had its adoption during the latter half of the previous decade. Nonetheless, the field is actively being studied, with the first relevant publications appearing in 2016 and a sharp increase thereafter. According to a systematic mapping study on the topic conducted in 2019 [15], most publications focused on executional aspects of different FaaS platforms. These mainly relate one way or another to the cold start problem and how it may be mitigated. Although the FaaS model may be the most common type of serverless computing, it is not the only form [16].

Security is another major topic in serverless literature. Many of the studies relate to how serverless applications may securely be executed in the public cloud. The nature of serverless places more responsibility on the cloud vendor regarding security, as they handle the operational and executional aspects of the applications. As the execution and scaling of serverless applications occur in a multi-tenant environment, it is vital that the cloud providers keep everything separate. Furthermore, serverless applications commonly depend on other cloud services offered by the same vendor. Communication between these different services often happens internally within the cloud infrastructure. As the infrastructure itself and many of the technologies are proprietary, external security researchers and experts cannot

adequately scrutinise the security of the platform [17].

Nevertheless, developers still have a prominent role in serverless application security. One study notes that the event-driven nature of serverless may increase the attack surface of applications implemented with the technology. As serverless applications often consist of multiple functions, a misconfiguration in one function trigger may compromise the application if the attacker can start to manipulate the execution of the workflow [18].

Many open questions related to serverless technologies and their usage still exist, including best use cases and design patterns. A recent study systematically collected and analysed several serverless applications from different sources to find community consensus [19]. The study presented noteworthy findings about the current state of serverless applications. One of their findings was that the primary motivators for building serverless applications were reduced running costs when IaaS resources are underutilised, reduced operational overhead, and seamless dynamic scaling of resources. Faster development speed was also noted as being one motivator, although it was not as common. The study also noted that serverless applications are not limited to specific application types but can be used for a wide variety of tasks. However, most of the applications featured bursty workloads. Such applications are idle most of the time but generate large bursts of activity whenever they are executed.

Some also see that serverless technologies, at least in their current form, are a step backwards. One study [20] argues that FaaS platforms are at odds with dominant trends in modern computing, as they restrict the ability to work with data and distributed computing resources efficiently. They see that the autoscaling characteristic of serverless, which increases the number of computational resources as needed, does have its benefits in some scenarios, such as independent tasks. However, as serverless functions are executed within separated, isolated containers, any data they process must be "shipped to them" rather than having the functions operate on the data directly where it is being stored. Furthermore, the isolation prevents fine-grained communication between different functions, which is needed for efficient parallelism. The proprietary nature of serverless platforms also discourages open-source innovation and results in further vendor lock-in.

The migration of serverless applications from one platform to another has also been investigated [13]. The study experimented with migrating four different typical FaaS use case implementations from AWS Lambda to Azure Functions and IBM Cloud Functions. During the migration process, the study identified multiple issues related to vendor lock-in. Some of the issues could be mitigated by designing the

initial solution in a certain way. Others would potentially result in a so-called dead end, meaning that a new solution would have to be built from the ground up for the new platform.

Most of the issues stemmed from the FaaS solutions being dependent on other services offered by the cloud vendor, such as object storage. Even if the new vendor has a corresponding service, it may not have the same features, such as a particular trigger. If a function would depend on such a trigger, migrating it to the new cloud vendor's platform may require the business logic to be redesigned. Moreover, even if the alternative service would have the same set of features, the way the FaaS platform implements the various interactions may be completely different. The study highlights AWS S3 and Azure Blob Storage, comparable object storage services, as examples. AWS handles service-specific events from S3 and other services using a separate library the vendor has developed. This made the function code of the initial AWS implementation dependent on the data types featured in the library. Furthermore, the ways the FaaS platforms access objects from their corresponding object storage services also differ. In AWS, the S3 events only contain a reference to the object in the storage service. Thus, the function itself must include the logic to retrieve the object. Meanwhile, in Azure, an object may be directly accessed from the event triggering the function as input [13].

It is possible to overcome these differences, but problems may occur during the migration of more complex solutions. The study suggests that a possible solution to mitigate the problem is to separate the business logic into a custom library. Cloud vendor-specific code would then be written in functions handling the input and outputs to other services, while the business logic would be referenced from the custom library [13].

## 2.3 Spring Boot

The serverless platforms presented in chapter 3 of this thesis are compared against more conventional counterparts developed using Java in combination with Spring Boot. This combination was chosen as a reference due to Java being one of the most used programming languages in the software development industry [21], especially for enterprise application development [22]. Moreover, Spring is the most used framework for Java, with Spring Boot prevalent in server-side applications [23]. The Java programming language already contains many features that make development easier. Nevertheless, frameworks like Spring reduce the need for developers to program standard functionalities, simplifying the code and speeding up

the development process [24].

Spring [25] is an open-source project comprising multiple modules that aim to support Java application development. Spring Framework is one of these modules. It provides a comprehensive library of prebuilt functionality and other features which help developers create robust Java applications, making the framework widespread in enterprise environments. The framework is incredibly flexible, allowing developers to configure it precisely to support the needs of their application. However, setting up the configurations using annotations or separate XML files can be complicated and time-consuming [26][27].

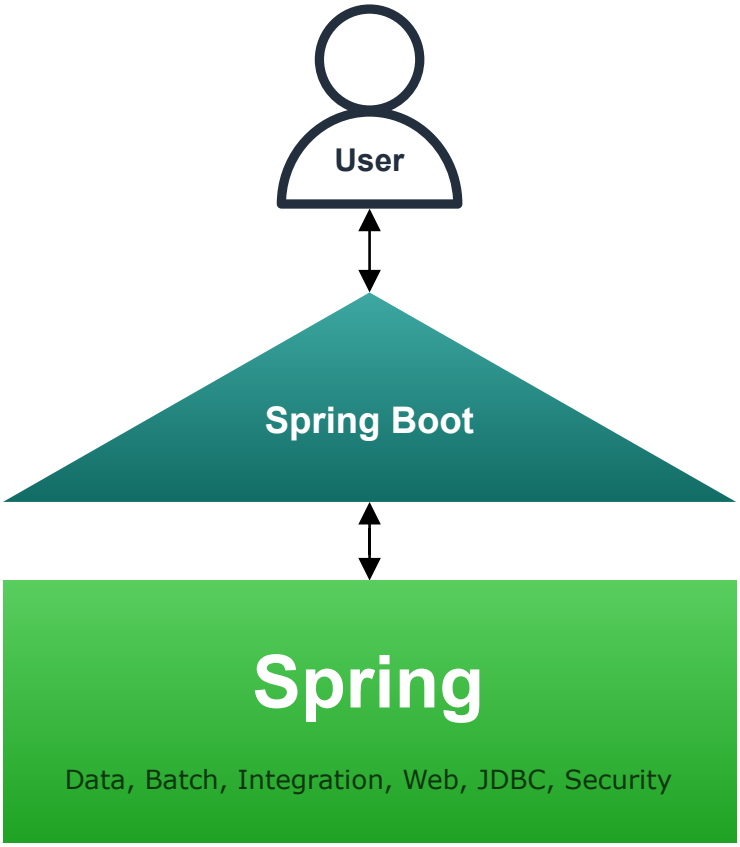


Figure 2.2: Spring Boot streamlines Spring development by making decisions for the user, making the framework more approachable [28]

Spring Boot addresses this complexity. It is another module part of the Spring project and can be seen as an extension to the Spring Framework. Spring Boot streamlines the development process by automating much of the configuration, resulting in the developers only needing to make minimal configurations or even eliminating the need altogether. This is done by having the Sprint Boot runtime make decisions for the developers by defaulting to certain implementations based on the classpath contents. This opinionated approach to configuration is known as auto-

configuration [24][27].

This auto-configuration can be well seen in the conventional implementations presented later in this thesis, which utilise Spring Boot. For example, in chapter 5.2.1, the implementation uses JPA for persistent storage. In the code itself, the repositories are only defined as interfaces. The JPA implementation that will be used during runtime is decided by Spring Boot, which automatically handles the injection of all dependencies.

# 3. Serverless in Azure

Azure is a cloud computing platform operated by Microsoft. Launched commercially in 2010 as Windows Azure [29], it has since become one of the industry's most prominent cloud vendors, only surpassed by Amazon's AWS [30]. Like other major cloud vendors, Azure operates in multiple regions, where they offer many different cloud computing services in several categories, such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). Azure's first serverless offering, Azure Functions, was launched in 2016 [31].

The serverless solutions presented in this thesis are developed for and using different serverless platforms offered by Azure. The used services are presented in further detail in this chapter.

## 3.1 Azure Functions

Azure Functions is Microsoft's Function-as-a-Service (FaaS) offering in Azure. As with other FaaS offerings, Azure Functions allows for the deployment and execution of pieces of code, known as functions, on their own without the need to consider the underlying platform running the code. Azure handles the provisioning and scaling of computing resources whenever needed as a function is executed [9][31].

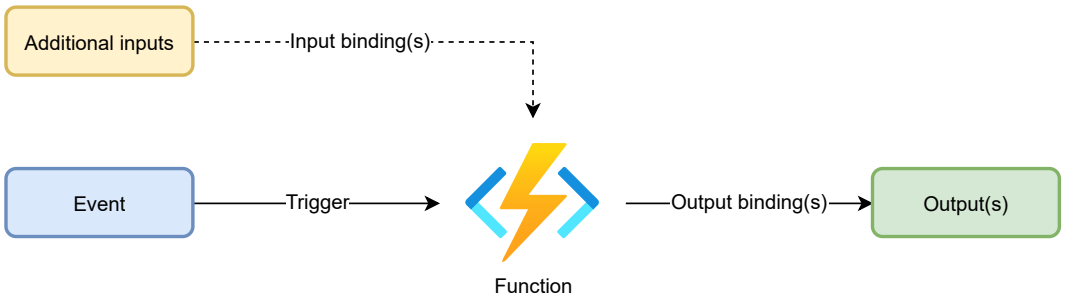


Figure 3.1: The core concepts of a function

Azure Functions is an event-driven platform, meaning that the execution of a function is always triggered in response to some event. The event itself often includes basic information, which acts as initial input for the function. Furthermore, additional input data may be retrieved from other sources as well. The inputs are

then used to produce one or more outputs [31].

Triggers, input bindings, and output bindings, which can be seen in figure 3.1, form core concepts in Azure Functions. They define when a function is to be executed, as well as how the data flows in and out of a function. Azure Functions contains several premade triggers and bindings, many of which integrate with Azure's other services. For example, it is possible to create a function that executes each time a new file is added to Blob storage, Azures object storage service. This is all done by adding and configuring a Blob storage trigger for the function. The inner workings of the trigger, which is responsible for detecting the added file and starting the execution of the function, are handled by Azure [32].

Azure Functions supports several programming languages, including C#, Java, JavaScript, and Python. Furthermore, package management systems, such as NuGet, npm and Maven, may be used for dependency management [9][33].

```java
public class Function {

    @FunctionName("HttpExample")
    public HttpResponseMessage run(
            @HttpTrigger(
                name = "req",
                methods = {HttpMethod.GET, HttpMethod.POST},
                authLevel = AuthorizationLevel.ANONYMOUS)
                HttpRequestMessage<Optional<String>> request,
            @QueueOutput(
                name = "msg",
                queueName = "outqueue",
                connection = "AzureWebJobsStorage")
                OutputBinding<String> msg,
            final ExecutionContext context) {
        context.getLogger().info("Java HTTP trigger processed a request.");

        // Parse query parameter
        final String name = request.getQueryParameters().get("name");

        if (name == null) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
              .body("Please pass a name on the query string")
              .build();
        }
        else {
            // Write message to message queue.
            msg.setValue(name);

            return request.createResponseBuilder(HttpStatus.OK)
              .body("Hello, " + name)
              .build();
        }
    }
}
```

Listing 3.1: An example of a function written in Java

How triggers and bindings are defined in a function varies depending on the programming language used. For compiled languages, such as Java, this is done by decorating the method declaration using special Azure Functions-specific annotations. An example of these annotations can be seen in Listing 3.1, where the first annotation defines the name of the function itself, the second the trigger of the function, and the third the output binding. The function tries to parse the “name” parameter from the request query provided by the trigger. A "Hello" message is returned if the parameter is present. If not, then HTTP response four hundred is returned [32].

In Azure Functions, individual functions are always deployed as part of a function app. A function app can be seen as a group of one or more related functions which run in the same runtime context. Therefore, they also share the same programming language, environment variables, and other application settings. Function apps are an important concept since they relate to how computational resources are scaled in the cloud. As functions are executed within a Function app, more computational resources will be allocated to the function app as a whole, not to individual functions. Consequently, functions within a Function app scale together, as they share the same computational resources. This should be considered when deciding which functions should be part of the same function app [31][34].

As of writing, function apps may be deployed in Azure using three types of hosting plans. The consumption plan is the default plan that follows the pay-as-you-go serverless model, as defined in the previous chapter. In this plan, function apps are billed based on a combination of memory usage and execution time, as well as the number of executions. The other two plans follow a different pricing model where at least some of the computational capacity is reserved for the user. As the user is paying for reserved capacity in these plans, they cannot be considered fully serverless. Moreover, function apps hosted on the App Service plan are deployed on a dedicated set of computing resources, removing the consumption aspect of serverless completely [9][11].

### 3.1.1 Durable Functions

In Azure Functions and FaaS in general, functions are generally stateless. This means that the functions will dissipate once they have performed their assigned task. This is fine for performing computational tasks that scale as needed but can make it difficult to implement more complex business logic, as this often requires multiple steps to be executed in coordination. Durable Functions is an extension to Azure Functions which solves this issue by making it possible to create orchestrated

stateful workflows [35].

The extension expands Azure Functions with four new function types. At the core is the Orchestrator function, which is responsible for the orchestration of the business logic. Orchestrator functions themselves do not perform any computations, as they solely define the steps that should be taken to perform the logic. These steps are instead performed by Activity functions, which the Orchestrator function calls. Activity functions are similar to regular Azure functions, which perform some computational tasks. Once an activity function has finished executing, it may return data to the orchestrator before dissipating. The orchestrator function may then pass this data on to any following activity functions. An example of this kind of workflow can be seen in figure 3.2. Note that the activity functions in the workflow may, using input and output bindings, exchange data from outside the workflow, as depicted in figure 3.1. The workflow is instigated by a so-called client function, which triggers the orchestration function using a particular type of binding [35][36].

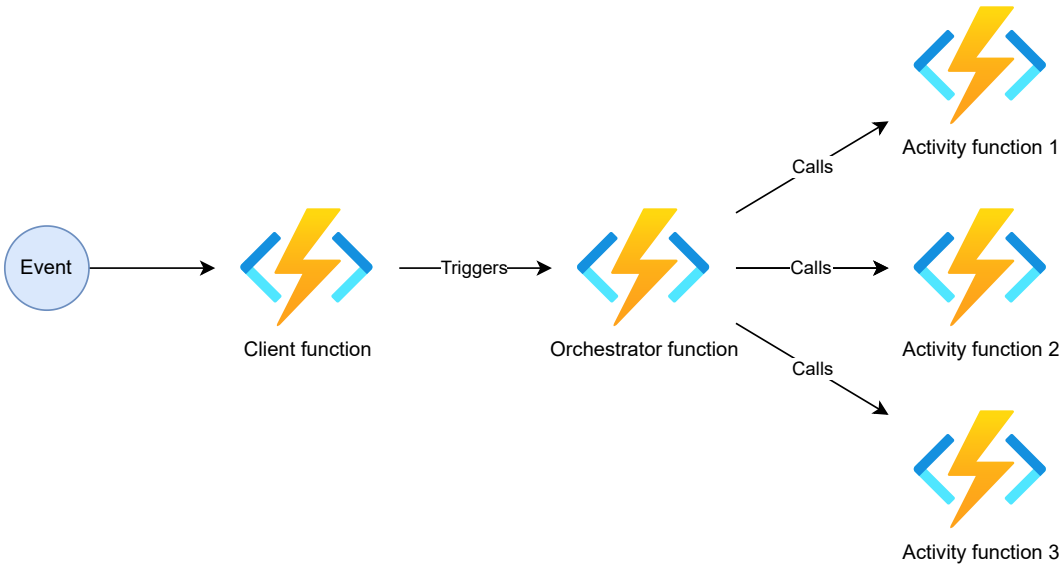


Figure 3.2: An example of function orchestration using Durable Functions

The Durable Functions extension works by storing the state of the orchestrator function in an Azure Storage account linked to the function app. The extension handles the inner workings of saving, retaining, and restoring the state of the orchestrator. This allows the function to resume even after prolonged pauses, which may occur in asynchronous processes [35].

## 3.2 Azure Logic Apps

Logic Apps is a serverless workflow engine in Azure. Unlike Azure Functions, which takes a "code-first" approach to implementing business logic, Logic Apps instead features its own visual designer tool where workflows are created. This "design-first" approach allows developers to create workflows that implement business logic without the need to write any code. As in other serverless platforms, the cloud provider handles the operative aspects of executing the workflow in the cloud as needed [9][37].

Azure offers two different resource types for Logic Apps, which are known as *Consumption* and *Standard*. The consumption type is the original version of Logic Apps, which utilises the serverless model described in chapter 2. The standard resource type is a newer variant of Logic Apps that runs in single-tenant environments, where the computational resources are reserved. As these two resource types have distinct runtimes, there are some minor differences between the two [38]. Therefore, as serverless is the subject of this thesis, the consumption type of Logic Apps will be described in this chapter. This version of Logic Apps is also used in the implementations described in the following chapters.

Logic Apps' workflows are comprised of connectors, which act as the building blocks of the business logic. Most connectors are essentially APIs that allow Logic Apps to communicate with other services. However, the APIs have been abstracted away from the users, leaving only a component that performs some specific operation. Workflows are created by adding and ordering connectors in a visual designer available in the Azure Portal. Connectors in Logic Apps can be divided into two types: triggers and actions [39].

Triggers are connectors that initiate the execution of a workflow. Therefore, each workflow must contain at least one trigger. As Logic Apps is an event-driven platform, each trigger is fired by some event. Triggers can be divided into two categories, polling and push types—the former works by regularly polling some endpoint at specific intervals for new pieces of data. If new data is available, the trigger initiates the execution of the workflow. The latter type of trigger listens to an endpoint and fires whenever an external source calls it. An example of a push type of trigger is shown in figure 3.3, where the workflow execution is triggered in response to an HTTP request [9][31][39].

Each trigger is followed by a set of actions. Actions are connectors of the workflow that are responsible for performing various tasks. Individual actions can, for instance, send an HTTP request to some endpoint, compose a JSON file, or trigger

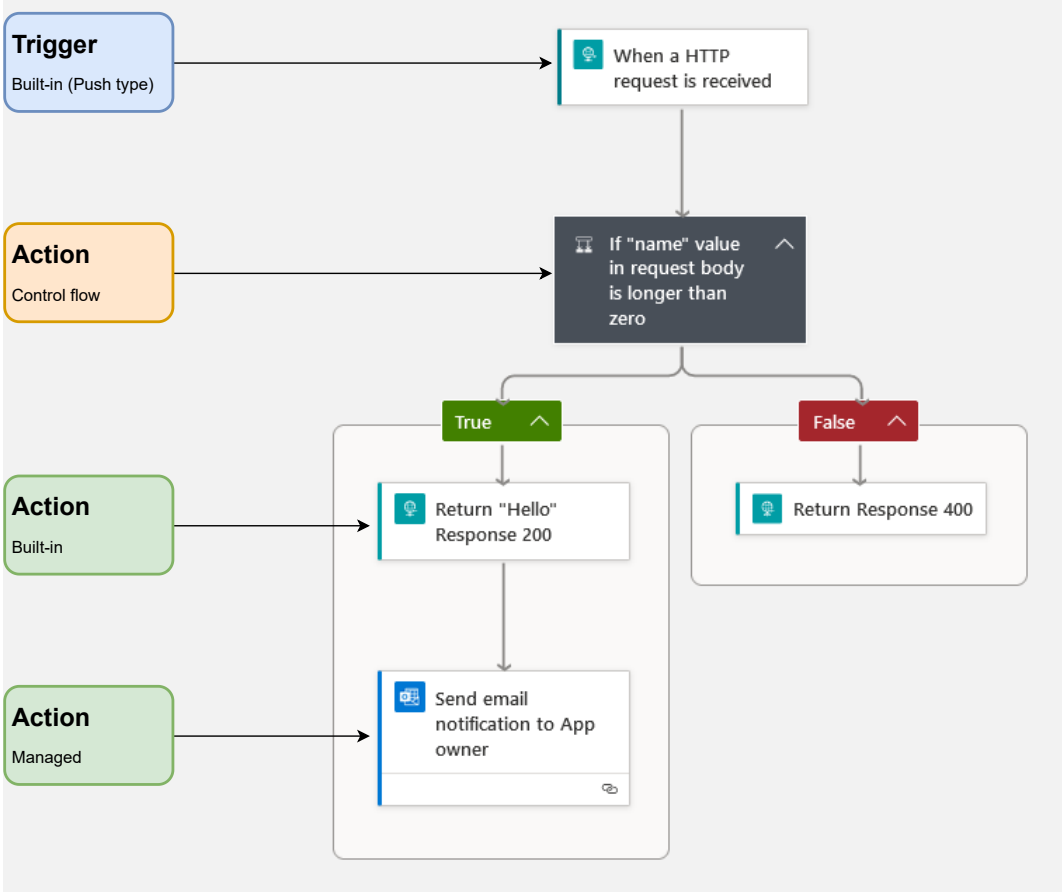


Figure 3.3: An example of a workflow in Logic Apps, with Triggers and Actions marked separately

a piece of code in Azure Functions to run. Logic Apps also contains a set of special actions which are used to alter the execution of the workflow during runtime. These are known as control flow actions. For instance, a control flow action may make the workflow execute some branch if a specific condition is met. Figure 3.3 depicts such a control flow action, where the following branch depends on the length of the "name" parameter [9].

Connectors in Logic Apps can be grouped into two categories: built-in and managed connectors. The former consists of connectors natively part of the workflow engine's runtime. This includes all control flow actions and many other basic actions and triggers, such as composing a JSON or sending an HTTP request. Some built-in connectors also allow for communication with other services within Azure, although this is somewhat limited in the consumption version of Logic Apps [39][40].

Managed connectors allow workflows to connect to other services within and outside of Azure. This group includes most of the connectors available in Logic

Apps, making it possible for workflows to respond and perform a wide variety of operations [41]. For example, a workflow could be triggered whenever a new email is received using a particular topic in Outlook [42], after which the contents of the message would be posted on a channel in Slack [43]. As these connectors are managed and maintained by the cloud provider, users do not have to consider the inner workings, such as background APIs, that perform the operations. The user only needs to configure the connectors and, depending on the connector, create a connection to the corresponding service. In the previous example, creating the connections means that the user only needs to sign in with their Outlook and Slack accounts and give Logic Apps the required access permissions. The authorisation tokens are managed and stored in Azure [39].

The differences between built-in and managed connectors are not that apparent to the users when designing workflows, except that they are located under different labels in the designer tool. However, they impact the workflow's operating costs, as they are priced differently. In the consumption plan, each connector execution is metered whenever a workflow runs. The number of connector executions is then multiplied by the unit price of that connector group. The unit cost of each group varies depending on the region the workflow is deployed on [44].

Furthermore, in the consumption version of Logic Apps, managed connectors are further divided into two groups: standard connectors and enterprise connectors. The latter contains all connectors related to enterprise systems, such as SAP, whilst the former contains all the rest. The only difference between the groups is the pricing [41][44].

### 3.2.1 Differences with Microsoft Power Automate

Microsoft also offers another workflow engine tool called Power Automate, which is similar to Logic Apps in many ways. Both allow for the orchestration of business logic in a similar-looking visual designer, and many of the available components are the same. Therefore, workflows in Power Automate and Logic Apps can, in some circumstances, look almost, if not completely, identical. Although the tools have many similarities, their intended use cases are somewhat different [45][46].

Power Automate was initially launched in 2016 with the name Microsoft Flow. The remnants of this old name can still be seen in spots, such as in the web address (*flow.microsoft.com*). The goal for Microsoft was to create a simple workflow engine for business users to automate different tasks. This was done by creating a lightweight version of Logic Apps, retaining much of the same design aspects of the service while removing more advanced features used by developers. Therefore,

it can be said that Power Automate is based on Logic Apps [31][47].

Some of the missing features in Power Automate that are available in Logic Apps include the ability to view and modify the JSON schema of the workflow, proper support for version control, and the availability of more advanced enterprise connectors. Furthermore, there are also differences in granular control. Workflows in Power automate are tied to and owned by individual users. Although multiple users can cooperate on flows, they will still be tied to their original creators. Conversely, in Logic Apps, workflows are resources in Azure owned by the tenant. This allows any user with the appropriate access rights to the resource to manage and make changes to the workflow. This also applies to any connector connections, which in Logic Apps are stored as resources in Azure. In Power Automate, connections are tied to the owner of the flow, or in the case of shared flows, the flow itself [41][46][48][49].

A notable differentiator is that Power Automate is not part of Azure. It belongs to Microsoft's Power Platform, a separate line of services with ties to Microsoft 365. Nevertheless, this does not mean that workflows created in Power Automate cannot interact with any services in Azure. On the contrary, it is even encouraged by Microsoft. However, the management of billing of Power Automate is done through different channels [50][51].

Power Automate being part of Microsoft's Power Platform also means that the service has different operating costs compared to Azure Logic Apps. Power Automate is a licence-based service, meaning that the user owning the workflow must have a licence. Access to Power Automate is included in Microsoft 365, allowing members of numerous organisations to utilise it to create workflows as part of that licence. However, the Power Automate licence included as part of Microsoft 365 has some limitations, such as limiting the user to a certain number of standard connectors. Access to more powerful premium connectors requires the user to acquire a separate licence. Various types of licences are available, some of which include more features, such as RPA functionality in desktop environments [50][51].

In summary, Power Automate targets end-users to help them automate their business processes, whereas Logic Apps is aimed at developers to create more advanced integrations and business workflows. Therefore, Logic Apps is the workflow engine used in the serverless implementation presented in chapter 5.

## 3.3 Azure Cosmos DB

Cosmos DB is a NoSQL database service in Azure. While it may not be serverless in the same way as defined in chapter 2, Cosmos DB works well with FaaS and other serverless services for several reasons. The database service is fully managed, meaning that the cloud provider is responsible for handling the operational aspects of the databases, such as scaling and maintenance. Furthermore, Cosmos DB features a consumption-based mode where throughput is scaled as needed, and the user only pays whenever resources are consumed. This consumption-based mode has therefore been dubbed serverless by Azure [52][53].

Cosmos DB provides different database APIs for working with the service. Regardless of the API used, the service will internally store the data in the same format, called Atom-Record-Sequence (ARS). However, the APIs allow applications to use the service similarly to existing database technologies. The native API for Cosmos DB is called Core and uses standard SQL syntax for database operations. Hence, it is also frequently called the "SQL API" or "Core (SQL)". This API allows many database operations to be performed similarly to relational databases, although there are some exceptions. While using the Core API, the data is stored as JSON documents. In addition to the Core API, Cosmos DB features compatibility APIs for other database technologies, such as MongoDB. However, for the purposes of this thesis, the Core API is utilised, as it is used by services such as Azure Functions [54][55][56].

The database API utilised is selected on the Cosmos DB account level. A Cosmos DB account is on the highest level in the element hierarchy and is directly created as a resource in Azure. Users can create one or more databases within an account. A database essentially acts as a group of containers, which are the elements that store data in the form of items along with other properties. This hierarchy is depicted in figure 3.4. The various elements in the hierarchy are represented differently depending on the database API used. For instance, the containers act as collections in the MongoDB API, whereas in the Core API, the containers are represented as is.

Resource usage in Cosmos DB is expressed in Request Units (RU), which is Azure's way of abstractly representing consumed computational resources. One RU is the equivalent of a one-point read of a one-kilobyte item. Each database operation has its own assigned RU cost, which varies depending on the database API and the size of the item being handled [57]. RUs are essential to Cosmos DB as they impact the performance and the costs associated with the service.

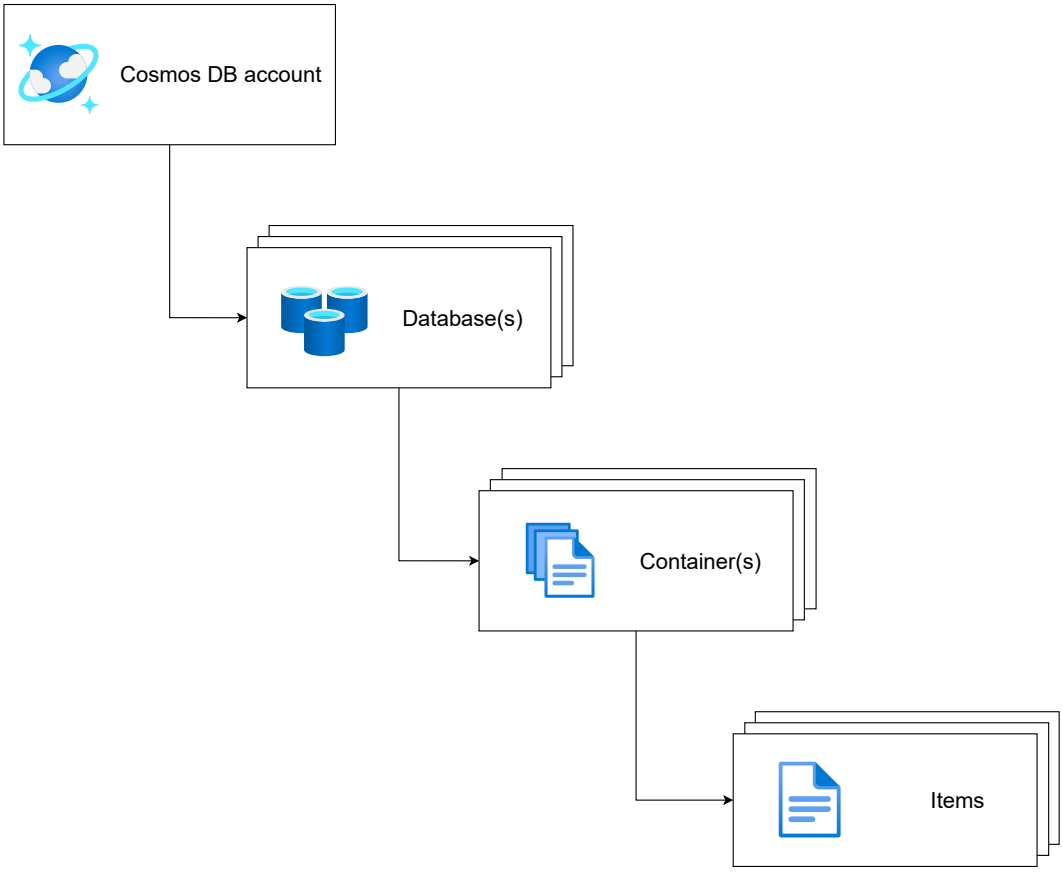


Figure 3.4: Simplified hierarchy of elements within Cosmos DB

Cosmos DB features two different capacity modes, provisioned throughput and serverless. In provisioned throughput, the user reserves a certain amount of throughput capacity from the service, expressed in Request Units per second (RU/s). Throughput capacity can be reserved for individual containers or the database as a whole. In the case of the latter, the capacity will be evenly distributed between the containers. The reserved capacity cannot be transcended, as the service will reject any exceeding database requests. Throughput capacity can either be provisioned manually or set to scale automatically. In the auto-scale mode, the user assigns a maximum throughput value for the database or container. The service will then, for each hour, bill the user for the peak throughput that occurred during that hour. However, this will always be at least 10% of the assigned maximum throughput, even if no resources are consumed. Therefore, the user will be paying for some reserved capacity even when using auto-scale. This is not an issue if the database is constantly receiving requests. However, if there are periods when the database is idle, the user will pay for reserved throughput capacity they are not using [57].

Serverless is Cosmos DB's consumption-based mode. In this mode, throughput cannot be provisioned. Instead, the user only pays for the number of RUs consumed

during the course of the month. This mode works well for applications that require persistent storage but only have infrequent traffic, as the user does not have to pay for unused reserved capacity. However, the serverless mode does feature some limitations not present in the provisioned throughput mode, such as a 50 GB storage limit and no geographical distribution. Furthermore, the serverless mode does not offer the same performance guarantees included in the provisioned throughput mode. Therefore, database operations may occasionally have higher latencies [53].

Both the provisioned throughput and serverless modes also feature costs for consumed storage. As of August 2022, the price is 0.25 USD per month for each gigabyte in the Europe-West region. In the case of provisioned throughput, the price is multiplied by the number of regions if geographical replication is on [58]. Nevertheless, unless large amounts of data are stored in the service for extended periods, the cost of consumed storage is negligible compared to reserved throughput or consumed RUs.

# 4. Presentation of the Use Cases

The use cases investigated in this thesis are presented in this chapter. Each use case is presented from the perspective of a fictional organisation known as *The Example Organisation* (*TEO*). TEO focuses on the business-to-business segment of the ICT market and offers its customers various services. Some services are produced in-house, while others are produced by a third party and resold to customers.

## 4.1 Use Case 1: Integration

TEO has decided to begin reselling a popular third-party software tool. The costs of the software are covered by a user-based licence, meaning that customers must have one licence for each individual user using the software. There are two types of licences available: *Standard* and *Pro*. The latter contains more features and is, consequently, more expensive. As the reseller, TEO is responsible for billing the resale value of the licences from its customers. The licences are to be billed at the beginning of each new month.

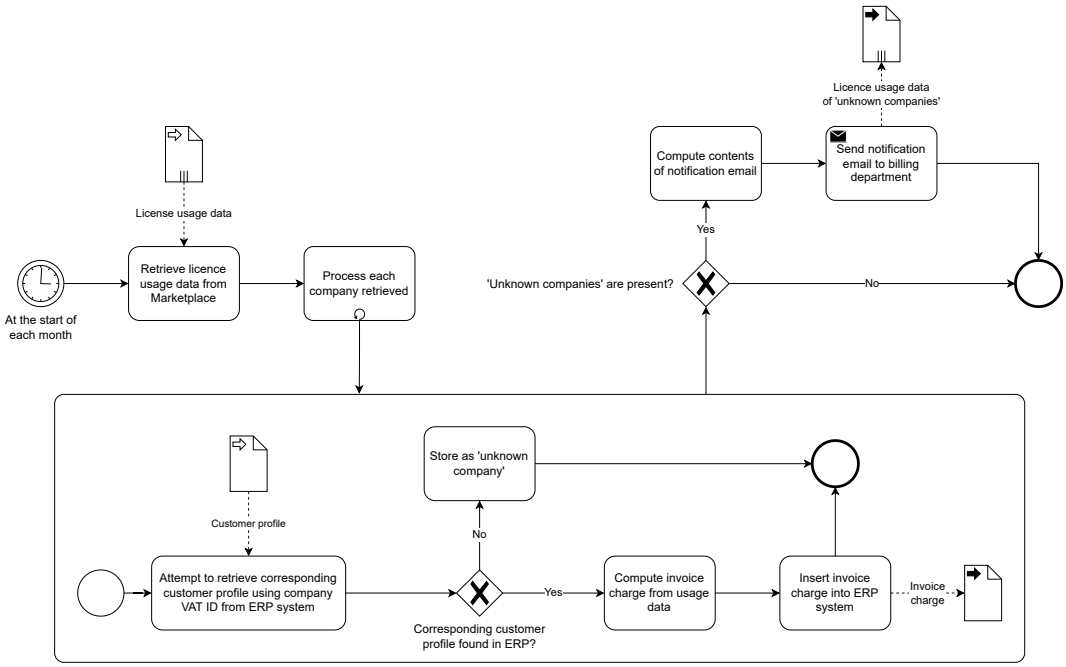


Figure 4.1: A BPMN diagram of Use Case 1

The licences are acquired and managed through a marketplace operated by the software vendor responsible for the tool. All licences within the marketplace must be registered to a company profile, which TEO creates for the customer upon ordering a licence for the first time. The company profile, among other things, contains the VAT ID of the customer. This acts as a reference between the customer's company profile in the marketplace and the customer's profile within the ERP system. Notably, these profiles can exist without one another. However, this causes problems for billing the licences, as a profile must exist in the ERP system for it to be possible to charge the customer.

The marketplace provides a REST API through which resellers may manage the tool licences. One of the API endpoints returns a complete list of all company profiles and the number of product licences they have been using during a particular month. As seen from listing 4.1, the list is returned in JSON format.

```
[
  {
    "companyName": "Example1 Corp.",
    "companyVatId": "FI1234567",
    "products": [
      {
        "productName": "Tool Licence Standard",
        "individualPrice": 5.50,
        "amount": 5
      },
      {
        "productName": "Tool Licence Pro",
        "individualPrice": 9.50,
        "amount": 2
      }
    ]
  },
  {
    "companyName": "Example2 Corp.",
    "companyVatId": "FI7654321",
    "products": [
      {
        "productName": "Tool Licence Standard",
        "individualPrice": 5.50,
        "amount": 3
      },
      {
        "productName": "Tool Licence Pro",
        "individualPrice": 9.50,
        "amount": 1
      }
    ]
  }
]
```

Listing 4.1: An example of licence usage data returned from the Marketplace

The ERP system already contains functionality to create and send invoices to customers. The invoice contents are based on invoice charges, which are records that may be inserted into the ERP system through various means. Each invoice-charge record contains the ID of the customer's profile in the ERP system, which acts as a reference. Additionally, the record also contains the total price, a description that appears on the invoice, and the period when the charge should be billed from the customer.

```
{
  "customerId": 12345,
  "total": 46.50,
  "description": "This is an description",
  "period": "2022-05"
}
```

Listing 4.2: An example of an invoice charge record

The ERP system features its own REST API, through which various operations may be performed. One of these endpoints allows external applications to insert new invoice charges into the system. The insertion is accomplished by sending an HTTP request containing the invoice charge as a JSON part of the request body. An example of such a JSON may be seen in listing 4.2.

For it to be possible for TEO to bill the product licences from the customers, a system integration must be implemented. A high-level depiction of of the needed integration may be seen in figure 4.2. Moreover, the process is covered in more detail in the BPMN diagram seen in figure 4.1. At the start of each new month, the integration should retrieve the total number of resold licences from the marketplace along with the company profiles they are associated with. Then, the integration should attempt to find the company's corresponding customer profile from the ERP system using the VAT ID retrieved from the marketplace. This is to check that the company has a customer profile in the ERP system. Moreover, the customer profile contains the customer ID of the company, which acts as a reference for the invoice charges.

If a company's corresponding profile is found in the ERP system, then the total cost of the product licences should be calculated, after which an invoice charge is to be generated and inserted into the ERP system. However, if no corresponding customer profile is to be found using the VAT ID, then the billing department should be notified of these companies by email. The email notification should contain the details of the companies and the product licences they have been using so that the billing department can take manual intervention to fix any possible issues.

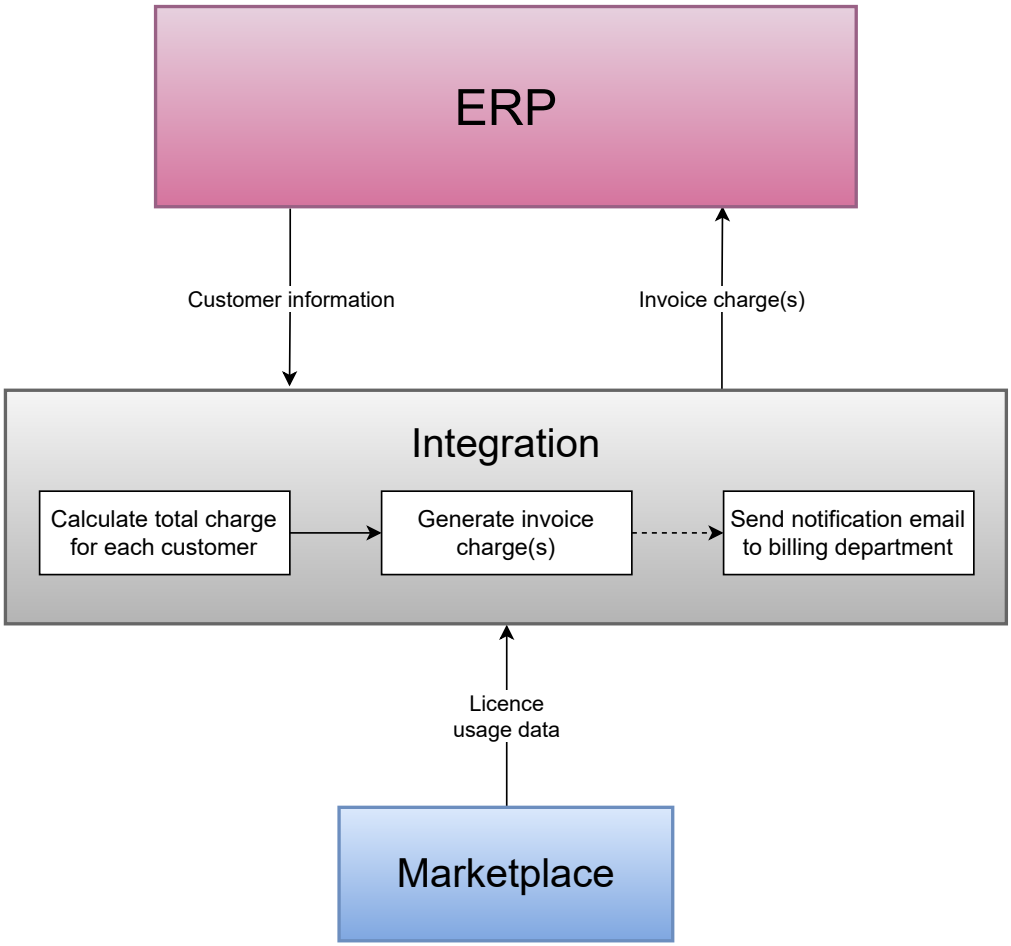


Figure 4.2: A high-level depiction of the integration and related systems

## 4.2 Use Case 2: Billing Engine

TEO has decided to start offering their in-house produced computing service. The service consists of three virtual computing instance plans: *Small*, *Medium*, and *Large*. The "bigger" the plan, the more computational resources it contains, thereby being more expensive. In addition, customers may also choose to attach additional storage to their plans for a premium. There are two storage options available: *Slow* and *Fast*, with the latter being more expensive than the former. The cost of the additional storage depends on the type of storage and the number of gigabytes the customer reserves, with a base cost for each gigabyte.

The computing instances have two types of operating systems available, an open-source variant and a proprietary version. The former OS is free and, as such, does not result in added costs. However, the proprietary OS requires an additional licence. The licence cost is added on top of the base price of the instance.

Computing instances and storage are both distinct types of *resources*. Both

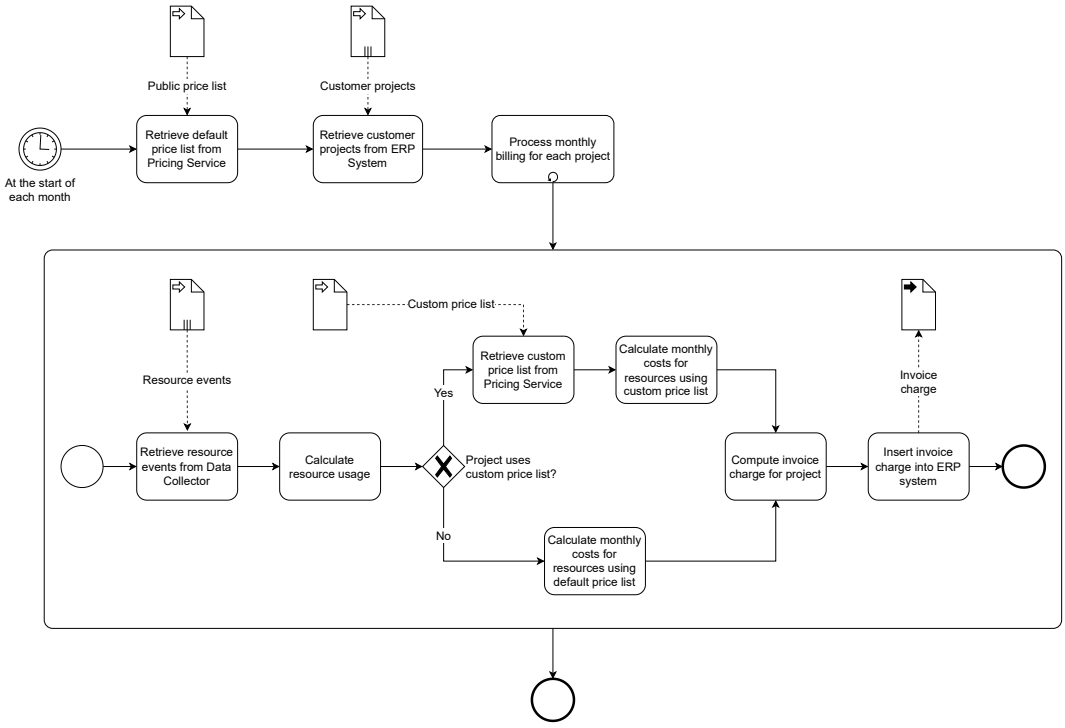


Figure 4.3: A BPMN diagram of Use Case 2

resource types are billed based on reserved capacity. Each resource has a *base price*, which is defined as the price of reserving the resource for one hour. Nevertheless, customers may create or delete computing instances and any attached storage at any time. At the end of every month, each resource is to be billed based on the time they have been reserved for the customer. The cost calculation is done by multiplying the hourly base price by the seconds the resource has been reserved, divided by 3600. Examples of this calculation for both resource types are shown in tables 4.1 and 4.2 respectively.

The base price for each resource is defined by a *price list*. By default, the base prices will be taken from the price list presented on the company website. This is known as the so-called *default price list*. However, customers may negotiate their own price lists with different base prices. These are known as *custom price lists*. Therefore, customers may be billed differently depending on the price list in use. Price lists are maintained in a separate *Pricing Service*, from where they can be retrieved as JSON objects. An example of such a JSON can be seen in listing 4.3.

On the computing platform, resources are always part of a *project*. A project is a group of related resources managed by a customer. Thus, one customer can have multiple projects. The platform itself is unaware of which project belongs to which customer. Instead, this information is stored in the ERP system as part of the customer's order data, which is created whenever a customer starts using the

| Instance Example | |
|---|---|
| Variant | Medium |
| Base price | €0.088/hr |
| Created at | 2022-05-05 10:00:00 |
| Deleted at | 2022-05-15 10:00:00 |
| OS Licence | Yes |
| Licence Cost | €0.032/hr |
| Cost calculation | |
| Usage | 10 days = 864 000 seconds |
| Cost | $\frac{(€0.088 + €0.032) * 864000s}{3600s} = €28.80$ |

Table 4.1: Cost calculation example for an computing instance

| Storage Example | |
|---|---|
| Variant | Fast |
| Base price/GB | €0.0005/hr |
| Created at | 2022-05-05 10:00:00 |
| Deleted at | 2022-05-15 10:00:00 |
| Amount | 100 GB |
| Cost calculation | |
| Usage | 10 days = 864 000 seconds |
| Cost | $\frac{(€0.0005/GB * 100GB) * 864000s}{3600s} = €12.00$ |

Table 4.2: Cost calculation example for storage

service. The ERP system allows the retrieval of all project records through its API. The records contain the IDs of the project and the associated customer. Moreover, the custom price list ID is also included, if applicable.

```
{
  "id": 1,
  "products":[
    {
      "productCode": "small",
      "price": 0.036
    },
    {
      "productCode": "medium",
      "price": 0.088
    },
    {
      "productCode": "large",
      "price": 0.145
```

```
    },
    {
      "productCode": "osLicence",
      "price": 0.032
    },
    {
      "productCode": "slow",
      "price": 0.0002
    },
    {
      "productCode": "fast",
      "price": 0.0005
    }
  ]
}
```

Listing 4.3: An example JSON price list from Pricing Service

The usage data of the different resources can be retrieved from a separate service called *Data Collector*. As the name suggests, the service gathers and stores data from the computing platform. One of the key pieces of data that Data Collector stores are the creation and deletion times of resources for each project. These are stored in the service as *resource events*, which may be retrieved using an API. The retrieved events are sorted based on the point in time at which they occurred.

```
[
  {
    "resourceId": 1000,
    "resourceType": "instance",
    "resourceVariant": "small",
    "eventType": "create",
    "happenedAt": "2022-05-05T10:00:10",
    "osLicence": false
  },
  {
    "resourceId": 1001,
    "resourceType": "storage",
    "resourceVariant": "fast",
    "eventType": "delete",
    "happenedAt": "2022-05-15T10:00:10",
    "units": 100
  }
]
```

Listing 4.4: An example of resource events retrieved from Data Collector

As seen in listing 4.4, the data stored in a resource event varies depending on the type of resource in question. However, standard parameters for each event are the resource ID, the resource type, the resource variant, the type of event in question (creation or deletion of a resource), and the point in time the event happened. Resource events for instances also contain a parameter on whether the created resource also requires a licence of the proprietary OS, while resource events for storage con-

tain the number of reserved gigabytes.

The resources are to be billed based on the events retrieved from Data Collector once a month. If a resource is created but not deleted during the billing period, it should be billed until the end of the period. Similarly, if a resource was deleted but not created during the billing period, it can be assumed to have been created during an earlier period. Therefore, it should be billed as if it was created during the start of the billing period.

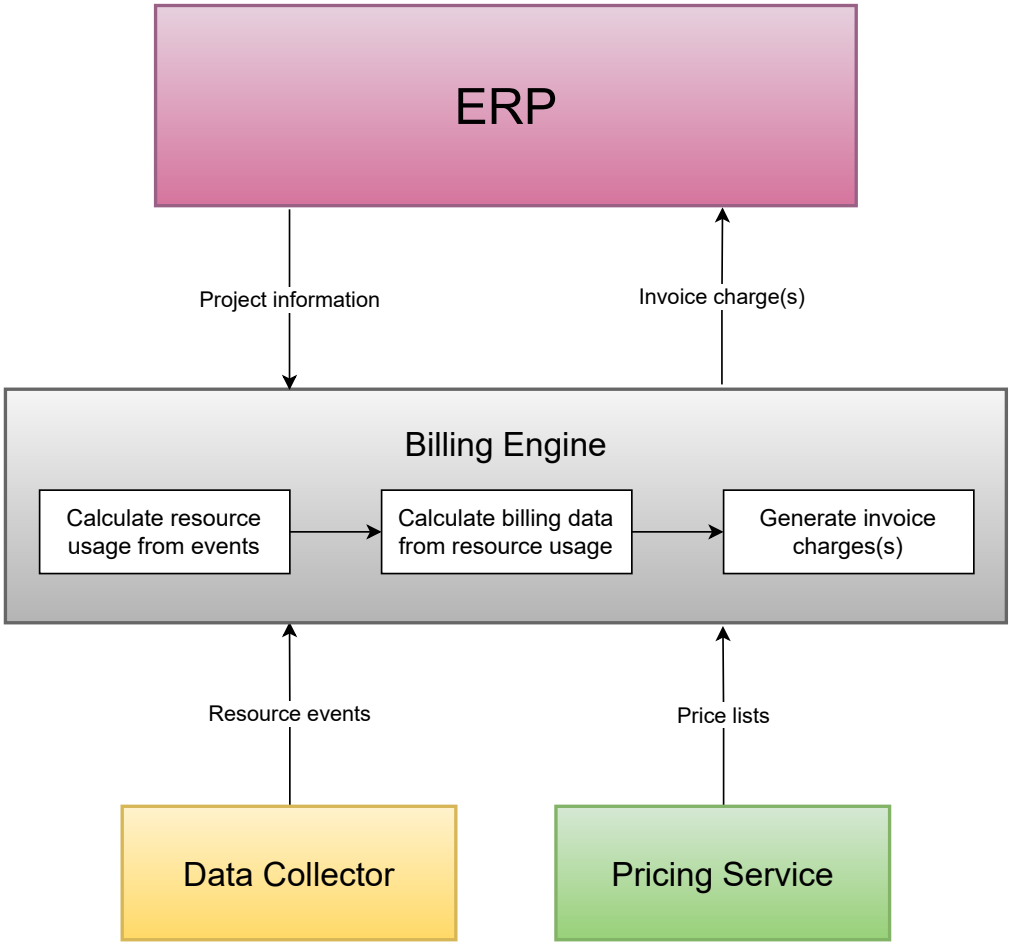


Figure 4.4: A high-level depiction of the Billing Engine and related systems

In production, the billing engine should automatically run once a month. A high-level depiction of the engine and the systems it is to interact with can be seen in figure 4.4. The billing engine should retrieve the required data from the three sources: the ERP system, Data Collector and Pricing Service, after which it should calculate the resource consumption and their charges. Finally, the results should be inserted into the ERP system as invoice charges. The invoice charge is the same as in Use Case 1, depicted in listing 4.2.

## 4.3 Use Case 3: API

Various teams within TEO create new customer profiles in the ERP system on a daily basis. The process involves entering numerous different details about a company into the profile in the ERP system, such as names, addresses, and phone numbers. This process can be somewhat tedious, as the user manually has to enter these details into the system to create the customer profile. Therefore, TEO has identified this as a possible case for automation.

The company details used to create the customer profiles in the ERP system are already publicly available. These are obtainable from the *Business Information System* (*BIS*), a public registry of companies maintained by a government agency. Furthermore, the system offers a public API through which the data may be retrieved. The API accepts the business ID of a company as a parameter and, if valid, returns all company information available in the system in the form of XML. Listing 4.5 contains an example of such a response.

```
<?xml version="1.0" encoding="UTF-8" ?>
<business id="1234567-8">
  <name>The Example Organization</name>
  <registrationDate>1970-01-01</registrationDate>
  <companyForm>Limited company</companyForm>
  <address>
    <street>Esimerkkitie 1</street>
    <postalCode>00000</postalCode>
    <city>Helsinki</city>
    <country>Finland</country>
  </address>
  <phone>+358 123456789</phone>
  <fax>+358 987654321</fax>
  <email>example@example.com</email>
</business>
```

Listing 4.5: An example of a response from the BIS API

The ERP system also contains its own API, which may be used by external applications to create new customer profiles in the system. The API accepts a JSON payload containing all the necessary company details required to create a new customer profile. If the operation is successful, the ERP system returns the customer ID of the new profile.

To automate the customer profile creation in the ERP system, TEO needs to implement a new internal API. The business process of the API is depicted in the BPMN diagram in figure 4.5. The new API should accept a business ID as a parameter. This ID should then be used to attempt to retrieve the details of the company from the Business Information System. If successful, the details should be mapped to the format accepted by the ERP system, after which it should be inserted. The

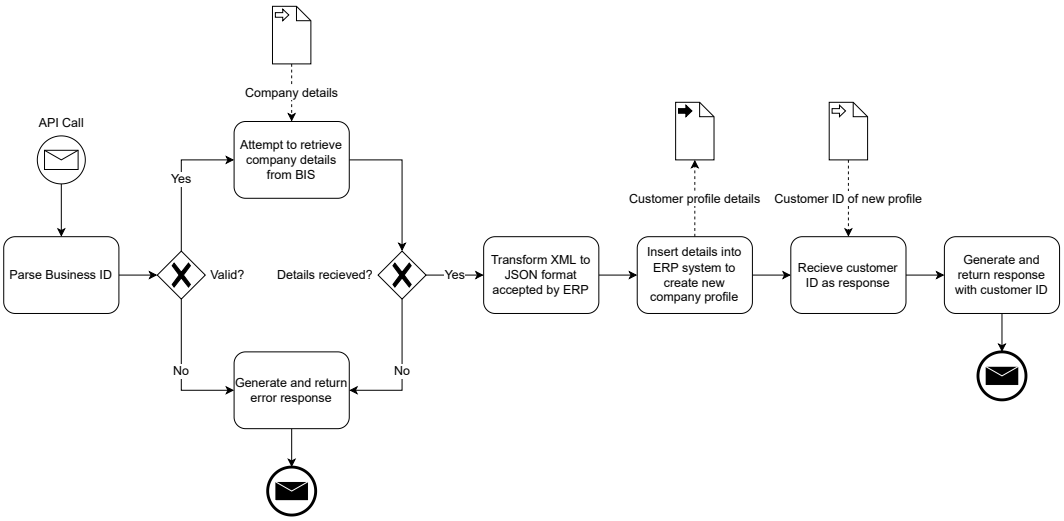


Figure 4.5: A BPMN diagram of Use Case 3

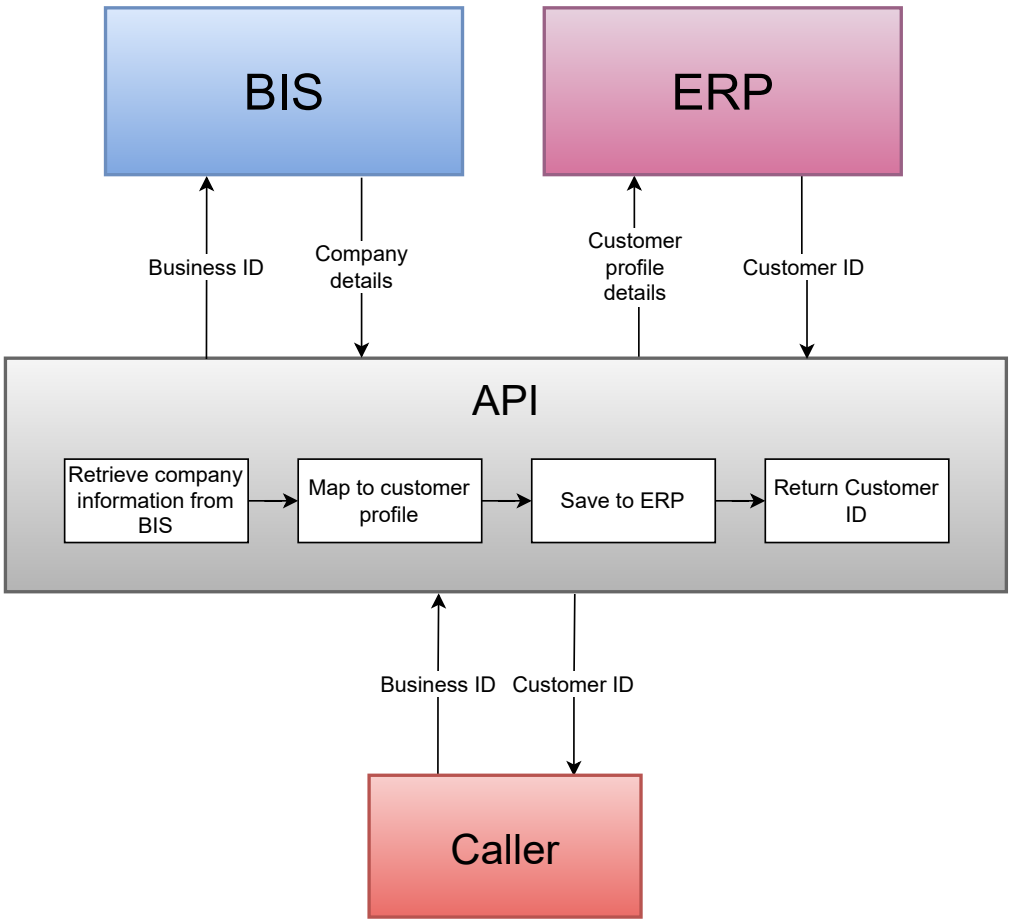


Figure 4.6: A high-level depiction of the API and related systems

new API should return the customer ID of the new profile in the ERP system if the operation is successful. The API would later be utilised by an internal tool suite used by various teams within TEO. However, this tool suite is not covered in this use case. Figure 4.6 shows a high-level depiction of the new API and the systems it is to interact with.

# 5. Implementations

This chapter presents implemented solutions for the three different example business use cases described in the previous chapter. The thesis presents two solutions for each use case. The first solution is implemented as a Spring Boot application programmed using Java, called the conventional implementation. The second solution is implemented using one or more serverless platforms in Azure, which were described in chapter 3. This is known as the serverless implementation. Both implementations and their respective development processes are later compared in chapter 6.

## 5.1 Use Case 1

### 5.1.1 Conventional Implementation

The application follows a design typical for Spring Boot applications. A complete class diagram of the solution can be seen in figure 5.1. The implementation primarily utilises Springs's own dependencies, along with Jackson for JSON operations. For logging, SLF4J is used together with Logback.

The primary business logic of the application is orchestrated in the *MarketPlaceIntegrationService* class. This logic is triggered by a scheduler method, depicted in listing 5.1. The method utilises Spring's scheduling functionality, which executes the method periodically on a specified date and time based on a *cron* expression. The expression is, in turn, fetched from a configuration file, making it possible to change the run intervals without modifying the application code itself. In addition to the scheduler, the business logic may be manually triggered using an HTTP request.

```
@Scheduled(cron = "${marketplace.cron.billing}", zone = "UTC")
public void runScheduledBilling() {
  LOGGER.info("Running scheduled marketplace billing");
  processMarketplaceBilling();
}
```

Listing 5.1: The scheduler method that initiates the business logic

The method *processMarketplaceBilling()* is responsible for executing the main business logic of the application. The method, which is depicted in listing 5.2, ini-

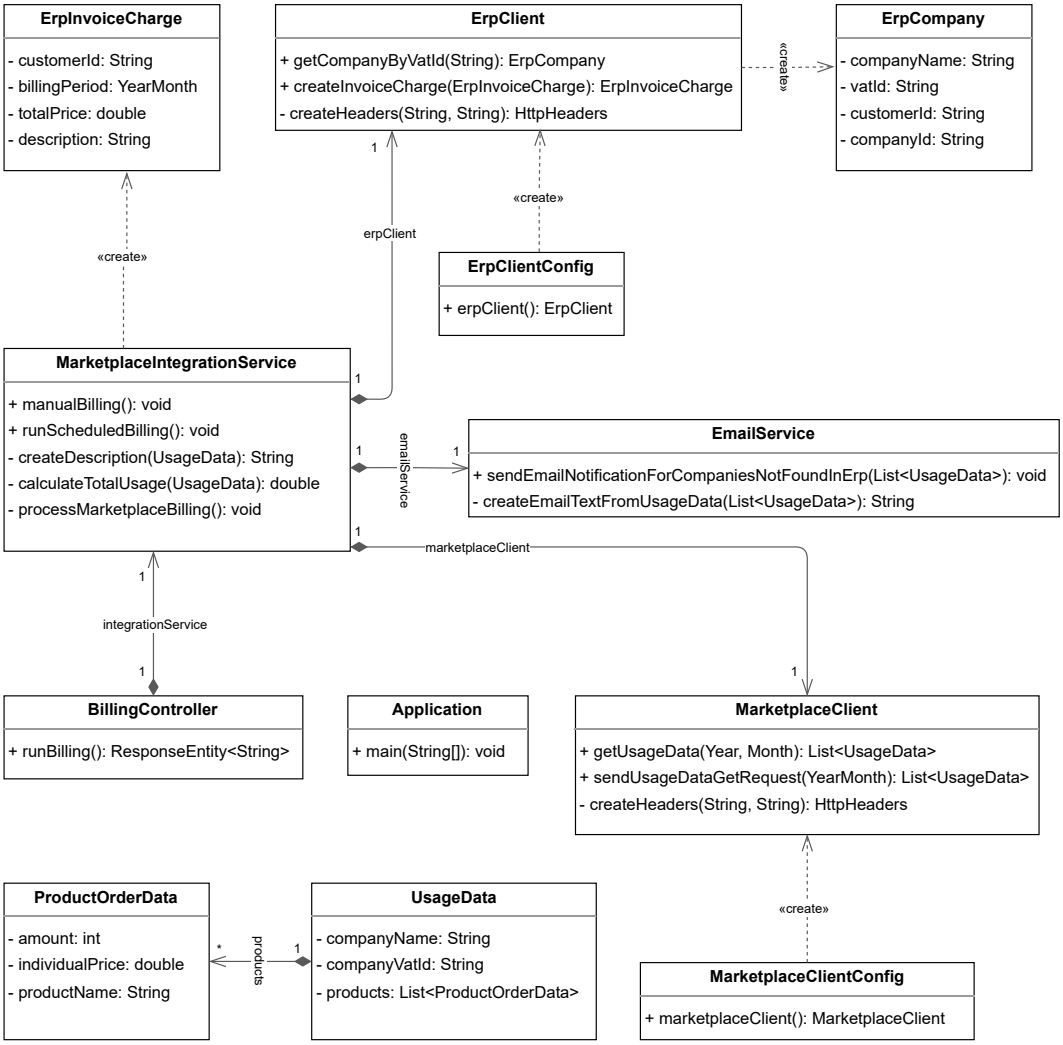


Figure 5.1: A class diagram of the conventional implementation of Use Case 1

tially attempts to retrieve the licence usage information from the Marketplace for the given period. If successful, the application then attempts to find the customer in the ERP system for each usage-data entry using the company's VAT ID. In case the company is found in the ERP system, then an invoice charge is generated and inserted into the ERP system for that customer. The total cost and description of the invoice charge are computed using support methods located in the MarketPlaceIntegrationService class. If the customer cannot be found using the given VAT ID in the ERP system, that entry is saved in a separate list. This list of so-called *unknown companies* is handled later at the end of the process.

Data exchange between the application, the Marketplace, and the ERP system is conducted through their respective clients. The clients contain methods that create the required HTTP requests to send and receive data in the form of JSON objects. Authorisation in the form of Basic authentication is also added as part of the re-

quest headers. This is all accomplished using a variety of web-related dependencies provided by Spring. The serialisation and deserialisation of the JSON objects are performed using *Jackson*.

```
private void processMarketplaceBilling(Year year, Month month) {
  try {
    List<UsageData> usageDataList = marketplaceClient.getUsageData(year, month
    );
    if (usageDataList.size() == 0) {
      LOGGER.warn("No usage data was found in marketplace");
      return;
    }
    LOGGER.info("Managed to retrieve {} data entries from marketplace",
        usageDataList.size());
    LOGGER.info("Starting usage data processing...");
    List<UsageData> entriesCustomerNotFoundInErp = new ArrayList<>();
    for (UsageData entry : usageDataList) {
      LOGGER.debug("Entry: {}, {}", entry.getCompanyName(), entry.
    getCompanyVatId());
      ErpCompany erpCompany = erpClient.getCompanyByVatId(entry.
    getCompanyVatId());
      if (erpCompany != null) {
        LOGGER.debug("Entry found in ERP using VatId: {}", entry.
    getCompanyVatId());
        createInvoiceCharge(entry, erpCompany);
      }
      else {
        LOGGER.info("Entry {} not found in ERP using VarId: {}",
            entry.getCompanyName(),
            entry.getCompanyVatId());
        entriesCustomerNotFoundInErp.add(entry);
      }
    }
    if (!entriesCustomerNotFoundInErp.isEmpty()) {
      emailService.sendEmailNotification(entriesCustomerNotFoundInErp);
    }
    LOGGER.info("Marketplace billing complete");
  } catch (Exception e) {
    LOGGER.error("Marketplace billing failure: {}", e.getMessage());
  }
}
```

Listing 5.2: The main processing logic of the conventional implementation in Use Case 1

The Spring beans for the clients are defined in their respective configuration classes, thereby allowing the Spring Inversion of Control (IoC) container to manage the creation and injection of these dependencies into other beans. Furthermore, the configuration classes insert the *URL*, *username* and *password* values from a separate configuration file. This makes it straightforward to deploy the application in different environments, such as testing or quality assurance.

At the end of the process, if one or several unknown companies are encountered,

then a notification email will be generated and sent to the billing department for them to be able to intervene manually. The email functionality is located in the *EmailService* class. The class contains logic to generate the contents of the email and send it to a list of recipients, which is defined in a configuration file. The email is sent using Spring's *JavaMailSender* class, which executes the sending via an SMTP server whose details are provided in the application configurations. This effectively allows for any mail server to be used, as long as all authorisation details are in order.

### 5.1.2 Serverless Implementation

Logic Apps was chosen as the primary development platform in Azure for the serverless implementation. One of the reasons for this choice was that Logic Apps is part of Azure Integration Services, a set of cloud services that Microsoft considers to address the core requirements of application integration [37]. As this use case aims to integrate a third-party marketplace and an ERP system, the development platform seems designed for this niche. Furthermore, the business logic in the use case itself is relatively straightforward, making it feasible to be orchestrated using a workflow engine in addition to more conventional programming methods. The workflow of the implemented Logic App is described below, along with figures.

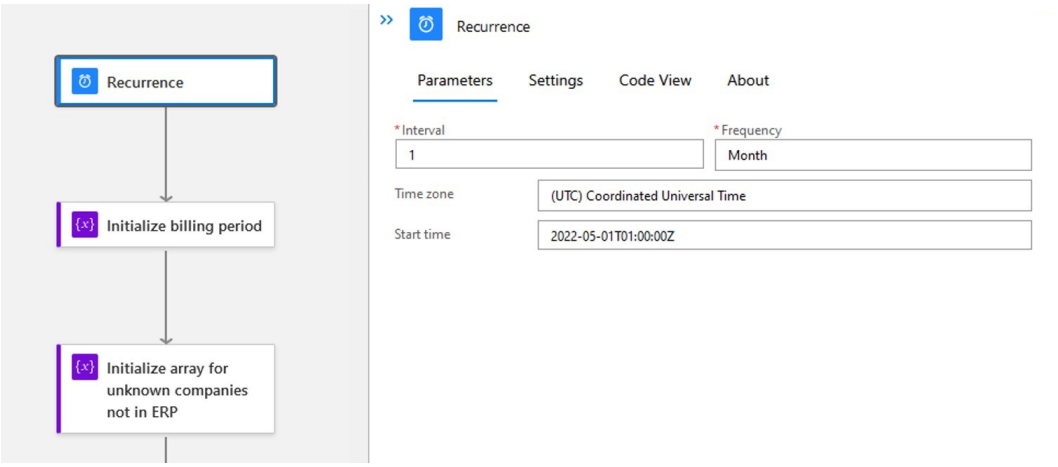


Figure 5.2: The trigger and initial actions of the workflow

The built-in *Recurrence trigger* initiates the execution of the workflow. The trigger accepts a few parameters, as seen in Figure 5.2. For this Logic App, the trigger is set to execute once at the start of each month at midnight UTC. Two variable-initialisation actions follow the trigger. These variables are primarily used later in the workflow's execution, but they are initialised at the beginning. This is

due to Logic Apps requiring variables to be initialised at the top level before any branching.

The former action initialises the current billing period, which is expressed as an ISO-8601 value containing the year and month of the period. The latter action initialises an empty JSON array, which will store licence usage data for unknown companies. Unknown companies are companies whose associated customer profiles cannot be found in the ERP system using the given VAT ID provided by the Marketplace.

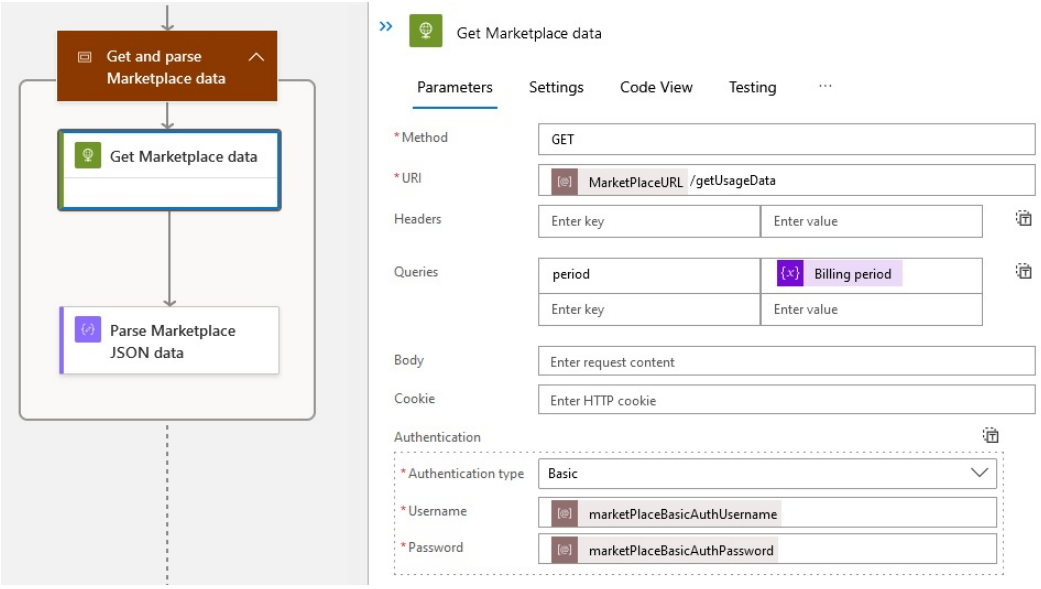


Figure 5.3: The retrieval and parsing of licence usage data from the Marketplace

After the variables have been initialised, the next step of the workflow retrieves the licence usage data from the Marketplace. The raw JSON data is first retrieved using the built-in *HTTP action*, which invokes the REST API of the Marketplace. Most of the parameters of the HTTP action, including the authentication credentials, are taken from the workflow's parameter section. How workflow-specific parameters are referenced can be seen in figure 5.3. These parameters may be set during deployment, allowing the Logic App to be deployed within different environments.

The raw JSON is parsed using the *Data Operation* action. The action allows for specific data values to be referenced later in the workflow's execution. Both the retrieval and parsing actions are performed within the same scope. This is to ensure that both actions pass successfully before proceeding to the following stage.

If at least one action within the scope fails, then the scope as a whole will also fail. Whether or not the scope succeeded is directly afterwards evaluated by a Conditional action, which can be seen in figure 5.4. If the scope fails, the Logic App will

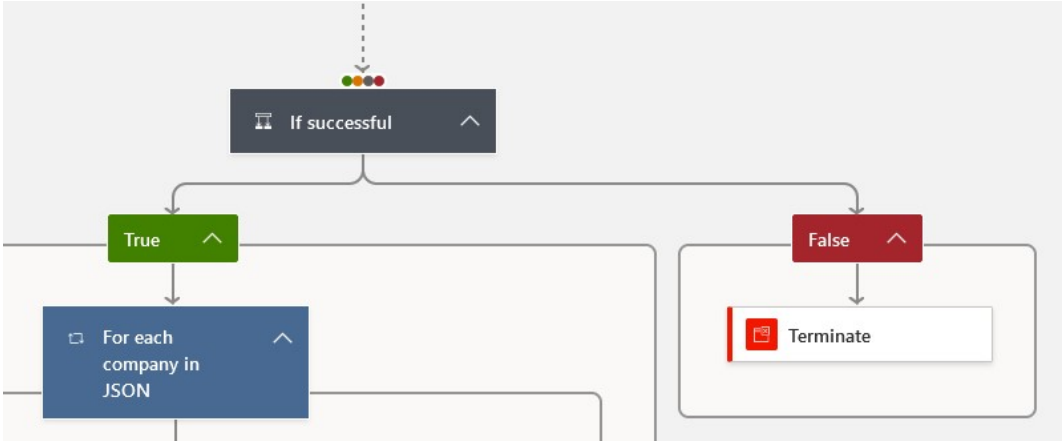


Figure 5.4: The results of the previous scope are evaluated

terminate using the built-in status code *Failed*. A monitoring alert may be created within Azure to notify of such failures.

If there are no failures in the scope, the workflow will enter a processing loop, depicted in figure 5.5. The loop processes each company in the retrieved licence usage data. First, the Logic App attempts to retrieve the company's customer profile from the ERP system using its VAT ID. The HTTP action, which invokes the REST API in the ERP system, is configured similarly to the action previously depicted in figure 5.3, only with different parameters. The Conditional action then evaluates the response of the HTTP request. The Logic App will parse the resulting JSON response if a customer profile is found in the ERP system. The parsing allows the values of the object to be individually referenced later. However, should no customer profile be found using the given VAT ID, then the company licence usage data will be appended to the array for unknown companies, which was initialised at the beginning of the workflow in figure 5.2.

The total cost and the description of the invoice charge are computed using Azure Functions. This is because the Logic Apps runtime can only perform rather simplistic computational operations using built-in actions and workflow definition expressions. The platform is unsuitable for conducting more advanced computations on data, or the operation will result in the workflow becoming overly complicated. Furthermore, this would result in an increased number of actions in the workflow, which increases the operational costs of running the Logic App. The topic of operational costs is covered in further detail in chapter 6.

The Logic App calls the custom Azure function and passes on the product information of the usage data to it as input parameters. The function then computes the total cost and description of the invoice charge, which is then returned as a JSON object to the workflow. The code is written in JavaScript as the Function App uses

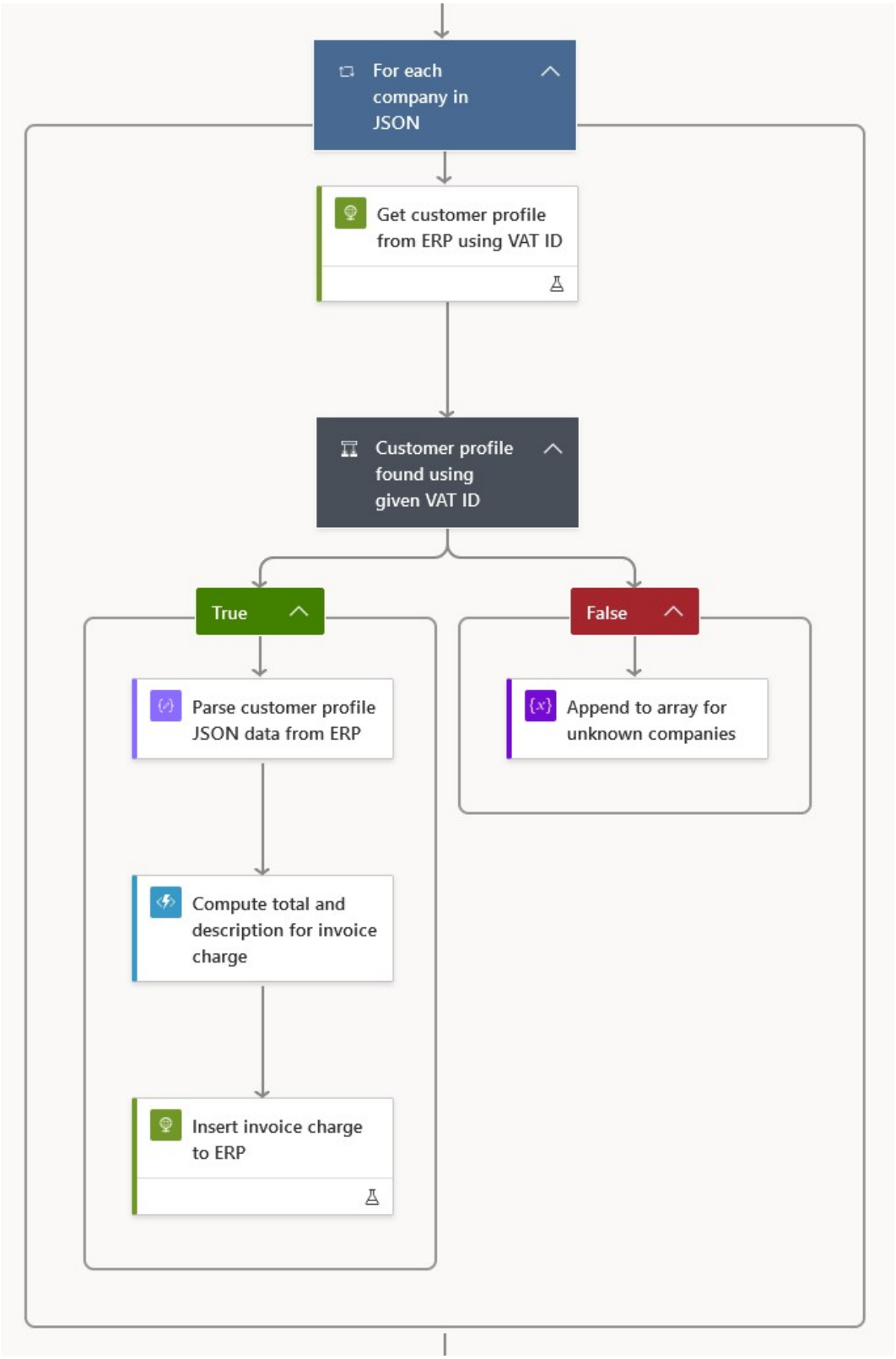


Figure 5.5: The main processing loop

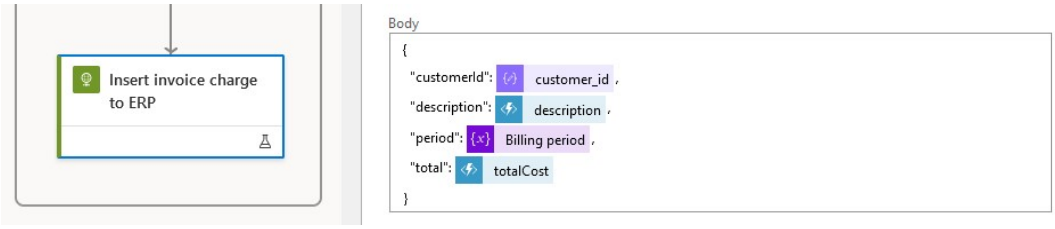


Figure 5.6: How the JSON of the invoice charge is constructed

the Node.js runtime. The invoice charge is saved to the ERP system using an HTTP action. The JSON object sent in the request's body is constructed within the HTTP action by referencing previous actions, as seen in figure 5.6. Notably, the values returned by the custom function are directly referenced using workflow definition expressions.

Once the main processing loop has ended, the workflow proceeds to one final conditional action, depicted in figure 5.7. The conditional action evaluates whether the array for *unknown companies* is empty or not. If the array is empty, the condition will evaluate to false, resulting in the Logic App terminating, as there are no more actions to follow. Otherwise, the workflow will continue, and an HTML table containing all the unknown companies and their licence usage data will be created. The *Data-Operation* action, which creates the HTML table, takes the JSON array initialised at the start of the workflow as input. The action can automatically detect the different properties of the JSON objects in the array and generate a table. This action is practical when data should be presented in a user-friendly format, such as in an email.

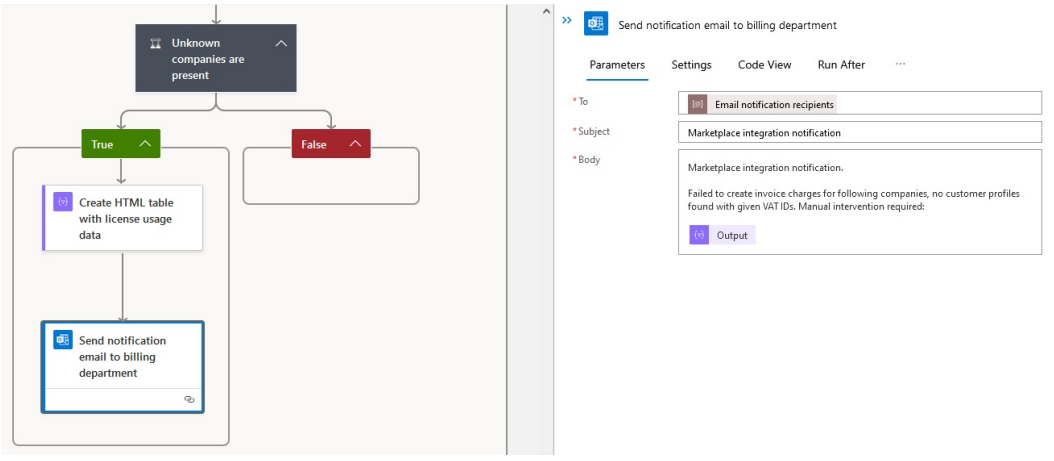


Figure 5.7: The final optional stage of sending a notification email

The email notification to the billing department is sent using the Outlook connector, a managed connector part of the Standard group. To utilise the action, the

developer must first create a connection between Logic Apps and the account from which the message is to be sent. Fortunately, this is a simple process, as the developer must only sign in with the account to be used and give the proper authorisations. The rest of the procedure, such as configuring the connection endpoints and storing the authorisation tokens, is handled by Azure.

The parameters of the Outlook connector may be entered into the designer once the connection has been established. As can be seen from figure 5.7, the recipients are configured to the connector as workflow parameters. This allows the list of recipients to be configured during deployment. The body of the email contains a hardcoded piece of text in addition to the HTML table created dynamically by the previous action. The Logic App will terminate once the Outlook connector has sent the email, as there are no further actions in the workflow.

## 5.2 Use Case 2

### 5.2.1 Conventional Implementation

Compared to the Spring application presented for the previous use case, this implementation is, in many ways, more complicated due to the more complex business logic. Nonetheless, there are many similarities in the design and the dependencies used. One significant addition is the introduction of persistence control with Jakarta Persistence API (JPA) and Spring Data JPA. A complete class diagram of the conventional implementation can is depicted in figure 5.8.

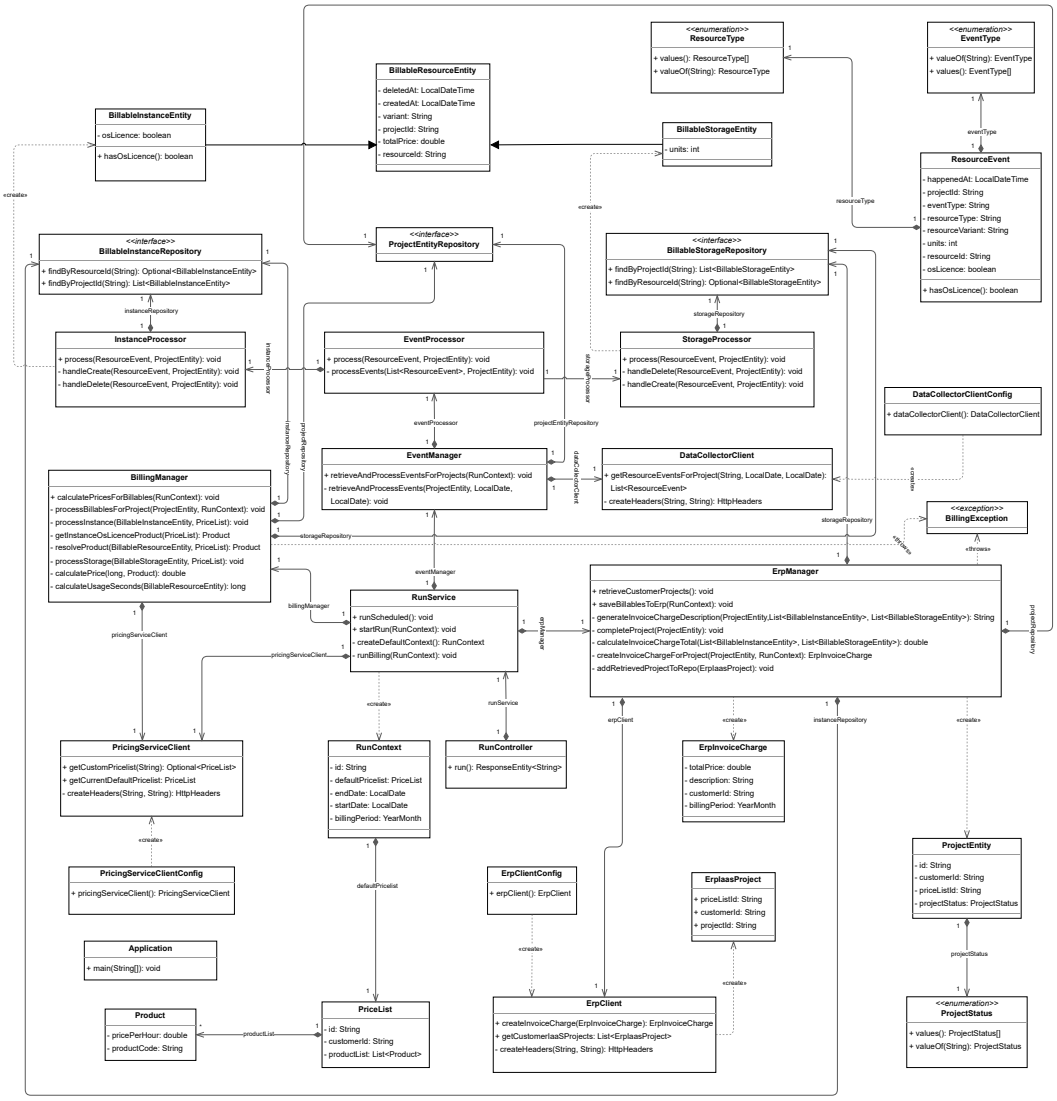


Figure 5.8: A class diagram of the conventional implementation of Use Case 2

The application utilises a context-based design pattern, in which the parameters and shared data of a billing run, such as the *start* and end *dates* and the *default price*

*list*, are stored in a specific context object. The object is then passed on to different stages in the billing process. The main business logic in the implementation is orchestrated within the *RunService* class using the *runBilling()* method. As seen in listing 5.3, the method itself is relatively simple, as the various stages of the billing process are delegated to separate classes. The start and completion of every stage in the process are logged and timed. The total duration of the billing run is also measured.

```
private void runBilling(RunContext context) {
  final Instant start = Instant.now();
  LOGGER.info("Beginning new billing run starting at {}, context is: {}",
    start, context);

  LOGGER.info("START - Retrieve customer projects from ERP");
  erpManager.retrieveCustomerProjects();
  LOGGER.info("COMPLETE - Retrieve customer projects from ERP");

  LOGGER.info("START - Retrieve and process resource events");
  eventManager.retrieveAndProcessEventsForProjects(context);
  LOGGER.info("COMPLETE - Retrieve and process resource events");

  LOGGER.info("START - Calculate prices for billables");
  billingManager.calculatePricesForBillables(context);
  LOGGER.info("COMPLETE - Calculate prices for billables");

  LOGGER.info("START - Save billables to ERP");
  erpManager.saveBillablesToErp(context);
  LOGGER.info("COMPLETE - Save billables to ERP");

  final Instant end = Instant.now();
  final long duration = Duration.between(start, end).toMillis();
  LOGGER.info("Billing run with contextId {} complete, duration {} seconds",
    context.getId(), duration/1000.0);
}
```

Listing 5.3: The method that orchestrates the main business logic of the billing-run, including the four stages

The billing process is divided into four stages, handled by three distinct manager classes: *ErpManager*, *EventManager*, and *BillingManager*. The ErpManager class is responsible for retrieving and inserting data to and from the ERP system. The EventManager class handles the retrieval and processing of all *resource events* from Data Collector. The BillingManager class is responsible for calculating the total cost of each resource. Common for all managers is that they process and store data in repositories. The primary reason is to save memory, as the number of resource events for each project can be substantial. Moreover, storing the data in a repository allows it to be retrieved both during the billing run and after it has finished.

In the first stage of the billing run, all current projects are retrieved from the ERP system using the *ErpClient* class. Each retrieved project is then mapped to a

*ProjectEntity*, which is then inserted into the *ProjectEntityRepository*. In the code itself, all repositories are only interfaces that extend the *JpaRepository* interface provided by Spring Data JPA. Spring Boot automatically configures and injects the actual database implementations based on dependencies in use by the application.

In the second stage, the *DataCollectorClient* class attempts to retrieve all resource events from Data Collector for the period defined by the context. This is repeated for each project in the ProjectEntityRepository. If the retrieval is successful, the list of events is passed on to the *EventProcessor* class. As the event data varies depending on the resource type in question, the events are further delegated to their respective processors, *InstanceProcessor* or *StorageProcessor*, as seen in listing 5.4.

```
1  public void processEvents(List<ResourceEvent> resourceEvents, ProjectEntity
       project) {
2    resourceEvents.forEach(resource -> process(resource, project));
3  }
4
5  private void process(ResourceEvent event, ProjectEntity project) {
6    LOGGER.debug("Processing event: {}", event.toString());
7    switch (event.getResourceType()) {
8      case INSTANCE:
9        instanceProcessor.process(event, project);
10       break;
11     case STORAGE:
12       storageProcessor.process(event, project);
13       break;
14     default:
15       LOGGER.error("Event {} for resource is of unknown resource type, could
       not be processed", event.getResourceId());
16   }
17 }
```

Listing 5.4: The processor used depends on the resource type

The event processing logic in the InstanceProcessor and StorageProcessor classes is similar, although there are slight variations. A snippet of the logic in the InstanceProcessor class is shown in listing 5.5. The handler method utilised for processing varies depending on the type of event. Regardless of the event type, the result is inserted into the repository as a *billable*. A billable is essentially a resource that is to be billed from the customer. For events with the type *create*, the handler method checks the repository if a resource with the same ID already exists. If none is found, a new billable will be created using the event data before saving it in the repository. At this point, the billable only contains a creation date. If a *delete* event is later processed for the same resource, then the *handleDelete()* method will save the deletion time to the billable.

```java
public void process(ResourceEvent event, ProjectEntity project) {
  if (!event.getResourceType().equals(ResourceType.INSTANCE)) {
    LOGGER.error("Resource event with id {} is not of INSTANCE type!", event.
    getResourceId());
  }
  switch (event.getEventType()) {
    case CREATE:
      handleCreate(event, project);
      break;
    case DELETE:
      handleDelete(event, project);
      break;
    default:
      LOGGER.error("Could not handle event of of type {} from resource {}",
    event.getEventType(), event.getResourceId());
      break;
  }
}

private void handleCreate(ResourceEvent event, ProjectEntity project) {
  Optional<BillableInstanceEntity> existingInstance = instanceRepository.
    findByResourceId(event.getResourceId());
  if (existingInstance.isPresent()) {
    LOGGER.error("Already found existing resource with same id: {}", event.
    getResourceId());
    return;
  }
  BillableInstanceEntity instance = new BillableInstanceEntity();
  instance.setCreatedAt(event.getHappenedAt());
  instance.setVariant(event.getResourceVariant());
  instance.setProjectId(project.getId());
  instance.setResourceId(event.getResourceId());
  instance.setOsLicence(event.hasOsLicence());
  LOGGER.debug("Saving new billable instance to repository: {}", instance);
  instanceRepository.save(instance);
}
```

Listing 5.5: Part of the event processing logic in the InstanceProcessor class

Once all the events have been processed for each project, the repositories should contain billables for all resources to be billed during this run. However, the billables at this stage are still missing one crucial parameter, the total charge. This is calculated and updated for each billable in the third stage of the billing run, performed by the *BillingManager* class.

The method *processBillablesForProject()*, shown in listing 5.6, processes each project individually. Firstly, the method checks whether the project uses a *custom price list*. If such a price list is not defined or cannot be retrieved, the default price list will be used instead. All billables for the project will then be retrieved from the repositories and processed by resource-type-specific processing methods. If an error occurs during the cost calculation process, the project's status in the repository will be set to *failed*.

```java
private void processBillablesForProject(ProjectEntity project, RunContext
    context) {
  LOGGER.debug("Starting to process billables for project {}", project.getId()
    );
  Optional<PriceList> priceListOptional = Optional.empty();
  if (project.getPriceListId() != null) {
    priceListOptional = pricingServiceClient.getCustomPricelist(project.
    getPriceListId());
    if (!priceListOptional.isPresent()) {
      LOGGER.warn("Custom price list for project {} not found using given
    price list id {}, using default price list instead.",
          project.getId(), project.getPriceListId());
    }
  }
  PriceList projectPricelist = priceListOptional.orElseGet(context::
    getDefaultPricelist);

  List<BillableInstanceEntity> instances = instanceRepository.findByProjectId(
    project.getId());
  List<BillableStorageEntity> storages = storageRepository.findByProjectId(
    project.getId());

  instances.forEach(instance -> {
    try {
      processInstance(instance, projectPricelist);
    } catch (BillingException e) {
      LOGGER.error("Failed to process instance {}, setting project {} to
    FAILED state. Exception: {}",
          instance.getResourceId(), project.getId(), e.getMessage());
      project.setProjectStatus(ProjectStatus.FAILED);
      projectRepository.save(project);
    }
  });
  storages.forEach(storage -> {
    try {
      processStorage(storage, projectPricelist);
    } catch (BillingException e) {
      LOGGER.error("Failed to process storage {}, setting project {} to FAILED
     state. Exception: {}",
          storage.getResourceId(), project.getId(), e);
      project.setProjectStatus(ProjectStatus.FAILED);
      projectRepository.save(project);
    }
  });
}
```

Listing 5.6: The processing logic in the BillingManager class

As in the event processing stage, the cost calculation varies depending on the resource type of the billable. However, the base cost and usage calculations, shown in listing 5.7, are the same for both. Resource usage is calculated by measuring the duration between the resource's creation time and deletion time. If the billable only has a creation time set, then the resource is assumed to have been created before the current billing period. Therefore, usage will be measured as if the resource was

created at the start of the month. Likewise, if the deletion time is missing, usage will be measured as if the resource was deleted at the end of the month.

The base cost for a billable is calculated by multiplying the total usage seconds by the hourly base price of the resource, divided by the number of seconds in an hour. The sum is then rounded up to the nearest cent using the scaling functionality of Java's *BigDecimal* class. In the case of instances, the total charge of the billable is the base cost plus the cost of any licensed OS. In the case of storage, the total charge is the base cost multiplied by the number of gigabytes reserved. Once the total charge has been calculated for a billable, it is inserted into the repository.

```
1  private double calculateCost(long totalUsageSeconds, Product product) {
2    final BigDecimal bd = BigDecimal.valueOf(((totalUsageSeconds * product.
       getPricePerHour()) / 3600));
3    return bd.setScale(2, RoundingMode.HALF_UP).doubleValue();
4  }
5
6  private long calculateUsageSeconds(BillableResourceEntity resource) throws
       BillingException {
7    if (resource.getCreatedAt() == null && resource.getDeletedAt() == null) {
8      throw new BillingException("Creation and deletion times are missing for
       resource with id".concat(resource.getResourceId()));
9    }
10   if (resource.getCreatedAt() == null && resource.getDeletedAt() != null) {
11     LocalDateTime creationTime = YearMonth.from(resource.getDeletedAt()).atDay
       (1).atTime(0,0,0);
12     LOGGER.debug("Resource {} creation time missing, calculating usage from
       start of month at {}", resource.getResourceId(), creationTime);
13     resource.setCreatedAt(creationTime);
14   }
15   else if (resource.getCreatedAt() != null && resource.getDeletedAt() == null)
        {
16     LocalDateTime deletionTime = YearMonth.from(resource.getCreatedAt()).
       atEndOfMonth().atTime(23,59,59);
17     LOGGER.debug("Resource {} deleting time missing, calculating usage
       stopping at end of month at {}", resource.getResourceId(), deletionTime);
18     resource.setDeletedAt(deletionTime);
19   }
20   long totalUsageMillis = Duration.between(resource.getCreatedAt(),resource.
       getDeletedAt()).toMillis();
21   return (totalUsageMillis/1000);
22 }
```

Listing 5.7: The base cost calculation for billables in the BillingManager class

Once the costs have been calculated for all billables in each project, then the fourth and final stage of the billing run will commence. In this stage, the ErpManager class uses the billables to generate an *invoice charge* for every project, which is subsequently inserted into the ERP system. This stage is skipped if the project's state is set to *failed*. The invoice charge is the same record as in the previous use case. Once the invoice charge has been successfully inserted into the ERP system,

the project status will be set to *processed*. After the final stage has been completed, the billing run will conclude. A total execution time is also calculated for reference.

There are two ways to initiate a billing run. The first is to schedule the execution using a *cron* expression, similarly to the previous use case. This is intended to be used in production, with the billing engine scheduled to run at the start of each month. The alternative method is to initiate the execution manually by invoking the HTTP-trigger method located in the *RunController* class. Custom billing-context parameters may be given this way. If none are given, a default billing context will be used instead. In the default context, the start and end date of the billing run will be the first and last day of the previous month.

### 5.2.2 Serverless Implementation

Azure Functions combined with the Durable Functions extension was chosen as the primary development platform for the serverless implementation. As this use case features the processing of a large number of projects along with their resource events, the computations were most straightforward to implement using a FaaS platform. Azure Functions by itself is well equipped to perform the required computations of the billing engine. However, as the use case features somewhat complex business logic, the Durable Functions extension had to be added for it to be possible to create the required orchestrations, as many functions had to be performed in a specific order.

In addition to Azure Functions, Cosmos DB is used to store the billables both during and after the billing run. As in the conventional implementation, this was done for it to be possible to access the data afterwards in case manual intervention is needed. Cosmos DB was chosen as the database platform as it integrates well with Azure Functions. Moreover, Cosmos DB's consumption-based mode suits this use case well.

The functions have been written in the C# programming language. Initially, the author had intended to develop the functions using Java, as this language is also used in the conventional implementation. However, at the time of writing this thesis, Durable Functions had only just received support for Java. Therefore, some features were still in development. One of these missing features was the bindings for Cosmos DB. As the author wanted this database platform to be featured in the serverless implementation, the programming language had to be changed.

Fortunately, C# features a syntax similar to that of Java. Furthermore, Azure Functions-specific bindings are defined similarly in both languages by decorating the methods and parameters. In C#, attributes are used for decoration, whilst Java

uses annotations. These similarities were why the author decided on C# as the programming language of the serverless implementation. Nevertheless, the programming language is of little relevance, as any supported language could have theoretically been used, with the end solution being the same, apart from language-specific differences.

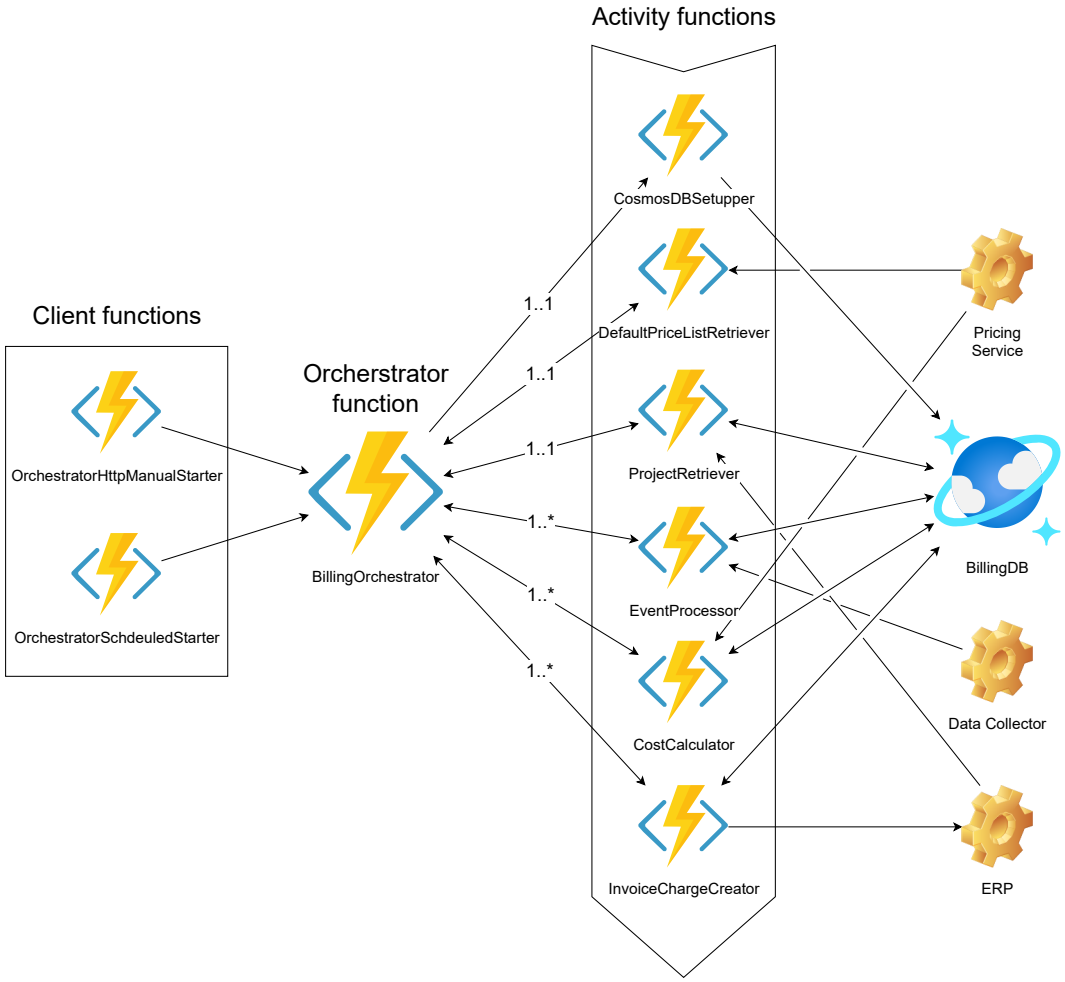


Figure 5.9: A diagram of the serverless implementation of Use Case 2

The solution consists of one orchestrator function and six activity functions. A high-level depiction of the implementation can be seen in figure 5.9. Each activity function is responsible for performing a particular step in the workflow. The name of the activity function gives a general idea of its role. The order in which the activity functions are executed in the orchestration can be seen from the arrow in the figure, with *CosmosDBSetupper* executing first. Furthermore, the implementation contains two client functions to control the orchestrator function.

The orchestrator function itself is relatively straightforward, as it mainly consists of activity function calls. As seen from listing 5.8, the orchestrator initially reads the

input and stores it in a variable *billingRunParameters*. This variable is then given to the activity functions in subsequent function calls. The variable is a *RunParams* type of object, which acts as a container for basic parameters used during the billing run.

```
[FunctionName("BillingOrchestrator")]
public static async Task<RunOutput> RunOrchestrator(
  [OrchestrationTrigger] IDurableOrchestrationContext context)
{
  var billingRunParameters = context.GetInput<RunParams>();

  /*
    Creates the database and containers in CosmosDB
  */
  context.SetCustomStatus("Setting up Cosmos DB for billing run");
  await context.CallActivityAsync("CosmosDBSetupper", billingRunParameters);

  /*
    Retrieves the current default price list from Pricing Service
  */
  context.SetCustomStatus("Retrieving default price list");
  var defaultPriceList = await context.CallActivityAsync<PriceList>("
    DefaultPriceListRetriever", billingRunParameters);

  /*
    Retrieves all Projects from ERP
  */
  context.SetCustomStatus("Retrieving project information");
  var projects = await context.CallActivityAsync<Project[]>("ProjectRetriever"
    , billingRunParameters);
```

Listing 5.8: The orchestrator function with the three first activity function calls

The first activity function to be called in the orchestration is CosmosDBSetupper. The function performs basic operations in Cosmos DB in preparation for the billing run. These operations include the creation of a new database along with the containers. As in the conventional implementation, the database is used to store all *projects* and *billables*.

Once the database has been set up, the following activity function to be called is *DefaultPriceListRetriever*. The function sends an HTTP request to Pricing Service to retrieve the current *default price list*. If successful, the function will return the price list to the orchestrator function to be stored in a variable. This variable is then later used during the cost calculation.

The third activity function invokes the API of the ERP system to retrieve all *projects* and their associated customer relationships, hence the name *ProjectRetriever*. These are then stored in their assigned container in Cosmos DB, similarly to how they are inserted into the repository in the conventional implementation. However, unlike the conventional implementation, the list of retrieved projects is

also returned to the orchestration as an array. This is because the following activity functions each process projects separately by taking the project to be processed as input. The orchestrator is responsible for assigning the projects to the activity functions. This allows for the parallel processing of the projects, which is one of the notable differences compared to the conventional implementation.

```
/*
  Retrieves and processes resource events to create billables in parallel
*/
context.SetCustomStatus("Retrieving and processing resource events");
var parallelEventProcessorTasks = new List<Task<Project>>();
foreach (var project in projects) {
  var eventProcessorTask = context.CallActivityAsync<Project>("EventProcessor"
    , (project, billingRunParameters));
  parallelEventProcessorTasks.Add(eventProcessorTask);
}
projects = await Task.WhenAll(parallelEventProcessorTasks);

/*
  Calculate costs for all billables in parallel
*/
context.SetCustomStatus("Calculating costs for billables");
var parallelCostCalculatorTasks = new List<Task<Project>>();
foreach (var project in projects) {
  var costCalculatorTask = context.CallActivityAsync<Project>("CostCalculator"
    , (project, billingRunParameters, defaultPriceList));
  parallelCostCalculatorTasks.Add(costCalculatorTask);
}
projects = await Task.WhenAll(parallelCostCalculatorTasks);

/*
  Create invoice charges and insert into ERP in parallel
*/
context.SetCustomStatus("Creating invoice charges");
var invoiceChargeCreatorTasks = new List<Task<Project>>();
foreach (var project in projects) {
  var invoiceChargeCreatorTask = context.CallActivityAsync<Project>("
    InvoiceChargeCreator", (project, billingRunParameters));
  invoiceChargeCreatorTasks.Add(invoiceChargeCreatorTask);
}
projects = await Task.WhenAll(invoiceChargeCreatorTasks);

var runOutput = GenerateOutput(projects);
context.SetCustomStatus("Billing run complete");
return runOutput;
}
```

Listing 5.9: The three following activity function calls of the orchestrator, which are executed in parallel

The start of the parallel processing of the projects occurs with the final three activity functions, seen in listing 5.9. For each step in the billing process, the orchestrator loops through the array of projects. Then, for each project, the orches-

trator calls the activity function with a tuple containing the project to be processed along with other parameters. These parameters vary slightly depending on the activity function in question. This type of function calling results in multiple activity functions being executed simultaneously, thereby improving performance.

The first parallel activity function to be executed is *EventProcessor*. The function is responsible for handling all tasks related to resource events. This includes the retrieval of all resource events for the project from Data Collector, in addition to the creation of the billables. The code of the EventProcessor function, shown in listing 5.10, follows a similar logic to that of the conventional implementation, apart from platform and programming language-specific differences. The function terminates by returning the existing project back to the orchestrator function. If something goes wrong while retrieving the resource events, the function will set the project's status to failed. The status will be updated to the Cosmos DB container before the project is returned to the orchestrator.

```
1  [FunctionName("EventProcessor")]
2  public async Task<Project> Run(
3    [ActivityTrigger] (Project project, RunParams runParams) inputs,
4    [CosmosDB(
5      Connection = "CosmosDBConnection")]
6      CosmosClient client,
7    ILogger logger)
8  {
9
10   Container projectContainer = client.GetDatabase(databaseName).GetContainer(
       projectsContainerName);
11   Container instanceContainer = client.GetDatabase(databaseName).GetContainer(
       billableInstancesContainerName);
12   Container storageContainer = client.GetDatabase(databaseName).GetContainer(
       billableStoragesContainerName);
13
14   /*
15     Retrieve resource events
16   */
17   List<ResourceEvent> resourceEvents;
18   try
19   {
20     resourceEvents = await RetrieveEvents(inputs.project.Id, inputs.runParams.
       StartDate.Value, inputs.runParams.EndDate.Value);
21   }
22   catch (Exception e)
23   {
24     logger.LogError($"Failed to retrieve resource events for project with id {
       inputs.project.Id}, setting status to FAILED. Exception: {e.Message}");
25     var project = inputs.project;
26     project.Status = ProjectStatus.FAILED;
27     await projectContainer.UpsertItemAsync<Project>(project);
28     return project;
29   }
30
```

```
31    /*
32      Process resource events
33    */
34    foreach (ResourceEvent e in resourceEvents) {
35      switch (e.ResourceType) {
36        case "instance":
37          await ProcessInstanceEvent(e, inputs.project.Id, instanceContainer,
      logger);
38          break;
39        case "storage":
40          await ProcessStorageEvent(e, inputs.project.Id, storageContainer,
      logger);
41          break;
42        default:
43          logger.LogError($"Could not handle resource event of type {e.
      ResourceType} from resource with ID {e.ResourceId}");
44          break;
45      }
46    }
47    return inputs.project;
48  }
```

Listing 5.10: The main body of the EventProcessor activity function

As seen from listing 5.10, the function uses a Cosmos DB binding to conduct database operations with the service. The binding is defined as a *CosmosClient* object in the function parameters. The parameter can be seen decorated with the CosmosDB-attribute, which also defines the connection string. In this case, that connection string is "CosmosDBConnection". The Azure Functions runtime's IoC-mechanism handles the injection of the CosmosClient and the logger whenever a new instance of the function is created.

Once all instances of the EventProcessor function have finished executing, the orchestrator function collects the returned projects as an array. The new array replaces the existing one, as seen previously in listing 5.9. This array is then used when the following *CostCalculator* functions are called. As the name suggests, the task of this activity function is to calculate the total cost of billables.

The CostCalculator function is responsible for calculating the total charges for the billables of a project. The initial stages of the function include a status check of the current project. If the status is anything but *unprocessed*, the function will instantly terminate, as there is no point in processing a project which encountered an exception during the previous stage in the billing process. As in EventProcessor, the function follows a similar logic to that of the conventional implementation, apart from platform and programming language-specific differences.

An example of a database query using the Core API of Cosmos DB is shown in listing 5.11. Although the Cosmos DB is a NoSQL database service, queries using the Core API are conducted using SQL. However, the iterators used to go

through the results of the query are specific to Cosmos DB, as they keep track of the continuation token specific to the service.

```
private async Task<List<T>> GetBillablesFromContainer<T>(string projectId,
    Container container) where T : BillableResource
{
  List<T> resources = new List<T>();

  QueryDefinition queryDefinition = new QueryDefinition(
    "SELECT * FROM items i where i.projectId = @searchterm")
    .WithParameter("@searchterm", projectId);

  using (FeedIterator<T> iterator = container.GetItemQueryIterator<T>(
    queryDefinition))
  {
    while (iterator.HasMoreResults)
    {
      FeedResponse<T> response = await iterator.ReadNextAsync();
      foreach (T resource in response)
      {
        resources.Add(resource);
      }
    }
  }
  return resources;
}
```

Listing 5.11: The logic to retrieve all billables from a container for a particular project

The final activity function of the orchestration is *InvoiceChargeCreator*. As the name suggests, the function is responsible for creating an invoice charge out of the billables for a project and inserting it into the ERP system. As seen from listing 5.12, the function initially checks the project's status and terminates unless the status is correct, similarly to the previous functions. After that, the function retrieves all billables for the project from Cosmos DB. The subsequent logic is similar to that of the conventional implementation. Once the invoice charge has been inserted into the ERP system, the project status will be set to processed before being returned to the orchestrator. In case of an exception, the status will be set to "failed".

```
[FunctionName("InvoiceChargeCreator")]
public async Task<Project> Run(
  [ActivityTrigger] (Project project, RunParams runParams) inputs,
  [CosmosDB(
    Connection = "CosmosDBConnection")]
    CosmosClient client,
  ILogger logger)
{
  var project = inputs.project;
  if (project.Status != ProjectStatus.UNPROCESSED)
  {
```

```
12      logger.LogWarning($"Project with id {project.Id} is in {project.Status}
          state and will be skipped");
13      return project;
14    }
15
16    Container projectsContainer = client.GetDatabase(databaseName).GetContainer(
          projectsContainerName);
17    Container instanceContainer = client.GetDatabase(databaseName).GetContainer(
          billableInstancesContainerName);
18    Container storageContainer = client.GetDatabase(databaseName).GetContainer(
          billableStoragesContainerName);
19
20    //Create and insert new invoice charge into ERP system
21    try
22    {
23      var instances = await GetBillablesFromContainer<BillableInstance>(project.
          Id, instanceContainer);
24      var storages = await GetBillablesFromContainer<BillableStorage>(project.Id
          , storageContainer);
25
26      ErpInvoiceCharge invoiceCharge = CreateInvoiceCharge(project, inputs.
          runParams.Period.Value, instances, storages);
27      await SaveInvoiceChargeToErp(invoiceCharge);
28
29      project.Status = ProjectStatus.PROCESSED;
30      await projectsContainer.UpsertItemAsync<Project>(project);
31      logger.LogInformation($"Project {project.Id} complete, status set as
          PROCSSED");
32    }
33    catch (Exception e)
34    {
35      logger.LogError($"Something went wrong creating invoice charge for project
           {project.Id}, setting project status to FAILED. Exception: {e.Message}");
36      project.Status = ProjectStatus.FAILED;
37      await projectsContainer.UpsertItemAsync<Project>(project);
38    }
39    return project;
40  }
```

Listing 5.12: The main body of the InvoiceChargeCreator function

Once the final instance of the *InvoiceChargeCreator* function activity function has finished executing, the orchestrator function will call the *GenerateOutput()* method, as seen in listing 5.9. The method returns a container object that includes the total number of projects retrieved and their current statuses. The orchestrator function then returns this object, thereby terminating. The termination of the orchestrator function thereby means that the workflow has come to an end.

A billing run may be started by triggering the orchestration, which can be done in two different ways. The client functions handle both of these methods. The first and primary method, which is intended to be used in production, is the one performed by *OrchestratorSchdeuledStarter*, depicted in listing 5.13. The function uses a timer trigger provided by Azure Functions, which starts the execution of the

function at a specific point in time, defined using a cron expression. The function triggers the orchestration function to run using a set of default parameters.

```
[FunctionName("OrchestratorSchdeuledStarter")]
public static async Task ScheduledStart(
  [TimerTrigger("%TimerSchedule%")] TimerInfo timer,
  [DurableClient] IDurableOrchestrationClient client,
  ILogger logger)
{
  var inputParams = GetDefaultParams();

  var id = await client.StartNewAsync("BillingOrchestrator", inputParams);
  logger.LogInformation($"Started orchestration with ID = '{id}'.");
}
```

Listing 5.13: The scheduled starter method

The alternative method is to start the orchestration manually using the *OrchestratorHttpManualStarter* function. This function exposes an API endpoint, which may be invoked with an HTTP Post-request. The request may include a JSON payload specifying the parameters of the billing run. If the request does not contain any parameters, then the default parameters will be used, which are the same as in the previous method.

The Durable Functions extension contains functionality to manage orchestrations using a built-in HTTP API [59]. One of the APIs may be used to retrieve the current status of an orchestration instance using its ID. The API returns a JSON response payload containing various information about the orchestration, such as input, output, and different timestamps. Listings 5.14 and 5.15 both show examples of such responses for this implementation. The orchestrator function changes the *customStatus* value whenever it proceeds to the next stage in the orchestrations, as seen in previous listings 5.8 and 5.9. In listing 5.15, the *output* is the object created by the *GenerateOutput()* method before the orchestrator function terminates.

```
{
    "name": "BillingOrchestrator",
    "instanceId": "add269d14efe46d9b94fb97da8f31d2a",
    "runtimeStatus": "Running",
    "input": {
        "start": "2022-07-01",
        "end": "2022-07-31",
        "period": "2022-08",
        "clearDatabase": true
    },
    "customStatus": "Retrieving and processing resource events",
    "output": null,
    "createdTime": "2022-08-02T11:32:29Z",
    "lastUpdatedTime": "2022-08-02T11:32:32Z"
}
```

Listing 5.14: An example of a status response while the orchestrator is running

```
{
    "name": "BillingOrchestrator",
    "instanceId": "add269d14efe46d9b94fb97da8f31d2a",
    "runtimeStatus": "Completed",
    "input": {
        "start": "2022-07-01",
        "end": "2022-07-31",
        "period": "2022-08",
        "clearDatabase": true
    },
    "customStatus": "Billing run complete",
    "output": {
        "projectsRetrieved": 1000,
        "processed": 1000,
        "failed": 0
    },
    "createdTime": "2022-08-02T11:32:29Z",
    "lastUpdatedTime": "2022-08-02T11:33:35Z"
}
```

Listing 5.15: An example of a status response once the orchestration has finished

## 5.3 Use Case 3

### 5.3.1 Conventional Implementation

The application follows a structure similar to the conventional solutions of the previous two use cases. In addition to many dependencies described in the previous use cases, this implementation features the use of Spring Security to add authorisation for the API. The application also features the use of JAXB to parse the retrieved XML data. A class diagram of the application is shown in figure 5.10.

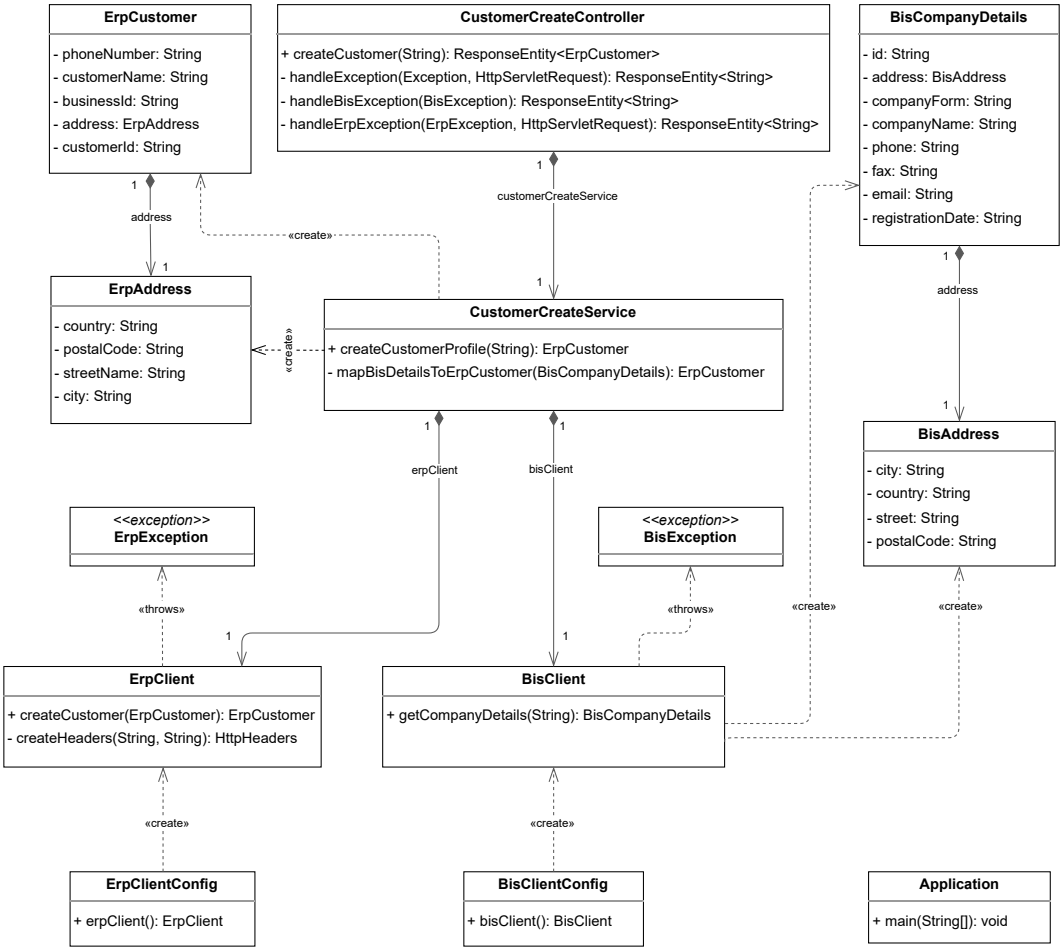


Figure 5.10: A class diagram of the conventional implementation of Use Case 3

The endpoint of the API is defined in the *CustomerCreateController* class. The class contains the method *createCustomerProfile()*, which defines the specifications of the endpoint. As seen from listing 5.16, this is done by annotations provided by Spring MVC. The method itself is simple, as it only passes on the given *businessId* parameter. The response returned varies depending on the result of the operation. If the operation is successful, the method will return an HTTP 201 response along

with the newly created customer profile as a JSON payload. However, an HTTP 404 response will be returned instead if no company details are found from the BIS using the given business ID. Separate exception handlers manage other responses in the same class.

```
@RequestMapping(value = "/createCustomer", method = RequestMethod.PUT, params
    = {"businessId"})
public ResponseEntity<ErpCustomer> createCustomer(@RequestParam(name = "
    businessId", required = true) String businessId) throws BisException,
    ErpException {
  LOGGER.debug("Called with business id: {}", businessId);
  Optional<ErpCustomer> response = customerCreateService.createCustomerProfile
    (businessId);
  return response.isPresent()
      ? new ResponseEntity<>(response.get(), HttpStatus.CREATED)
      : new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

Listing 5.16: The method defining the endpoint of the API

The *CustomerCreateService* class manages the rather simplistic business logic of the application. This is accomplished using the *createCustomerProfile()* method, which is called by the controller whenever the endpoint is triggered. As with the previous implementations, client classes are used for handling the data exchange between the application and other services. As seen from listing 5.17, the method first uses the *BisClient* to attempt to retrieve the company details. The details are returned in the form of a *BisCompanyDetails* object, which is modelled after the BIS response, as seen in listing 4.5. As the response is in the form of XML, the *BisCompanyDetails* class is decorated with JAXB-specific annotations for unmarshalling.

```
public Optional<ErpCustomer> createCustomerProfile(String businessId) throws
    BisException, ErpException {
  Optional<BisCompanyDetails> bisCompanyDetails = bisClient.getCompanyDetails(
    businessId);
  if (bisCompanyDetails.isPresent()) {
    ErpCustomer erpCustomerProfile = mapBisDetailsToErpCustomer(
    bisCompanyDetails.get());
    return erpClient.createCustomer(erpCustomerProfile);
  }
  return Optional.empty();
}
```

Listing 5.17: The createCustomerProfile() method in the CustomerCreateService class

If the company details are found for the given business ID in the BIS, then the relevant values of the BisCompanyDetails object are mapped to the format supported by the ERP system. The *ErpCustomer* class represents this format. As the object needs to be serialised into JSON to be inserted into the ERP system, the

class is decorated with Jackson-specific annotations. The insertion is performed by the *ErpClient*, which returns the created profile along with its customer ID upon successful insertion.

If either BisClient or ErpClient receives a response from their respective service, which is classified as an HTTP client or server error, then a custom exception will be thrown. These custom exceptions can be seen in the class diagram depicted in figure 5.10. The exceptions are thrown by the *createCustomerProfile()* method as well, as the exceptions are handled in the controller.

The controller features three exception-handler methods. These methods catch the thrown exception and create an appropriate response to be returned by the API. Examples of such handler methods can be seen in listing 5.18. As seen from the listing, the methods are decorated using the *@ExceptionHandler* annotation provided by Spring Web. The annotation states that the method should catch and handle the type of exception declared in the annotation. The HTTP status and message of the response returned vary depending on the values in the exception. The exception contains information about what caused the client to throw the exception.

```
1  @ExceptionHandler(BisException.class)
2  private ResponseEntity<String> handleBisException(BisException e) {
3    LOGGER.error("Failed to retrieve business info from BIS, status {}, message:
       {}", e.getHttpStatusCode(), e.getMessage());
4    return ResponseEntity.status(e.getHttpStatusCode().is4xxClientError() ?
       HttpStatus.INTERNAL_SERVER_ERROR : HttpStatus.BAD_GATEWAY).body("Failed to
       create customer profile, could not retrieve business info from BIS");
5  }
6  @ExceptionHandler(ErpException.class)
7  private ResponseEntity<String> handleErpException(ErpException e,
       HttpServletRequest request) {
8    if (e.getHttpStatusCode().equals(HttpStatus.CONFLICT)) {
9      LOGGER.info("Existing customer found in ERP with same business ID, no new
       profile created. {}", request.getQueryString());
10     return ResponseEntity.status(HttpStatus.CONFLICT).body("Existing customer
       profile already exists in ERP for given business id");
11   }
12   LOGGER.error("Failed to create new customer profile in ERP, status {},
       message: {}", e.getHttpStatusCode(), e.getMessage());
13   return ResponseEntity.status(e.getHttpStatusCode().is4xxClientError() ?
       HttpStatus.INTERNAL_SERVER_ERROR : HttpStatus.BAD_GATEWAY).body("Failed to
       create customer profile, could not save to ERP");
14 }
```

Listing 5.18: Two exception handlers in the in controller

The API is secured using Spring Security. Spring Boot automatically configures Spring Security as soon as the dependency has been added to the project. However, the auto-configurations by Spring Boot were not suitable for this solution and therefore had to be disabled. Thus, a separate *SecurityConfig* class had to be created, which contains the needed Spring Security configurations to secure the API.

### 5.3.2 Serverless Implementation

The serverless implementation solely uses Azure Function as the development platform. The FaaS platform meets the prerequisites for implementing a simple API. As the required business logic is somewhat straightforward means that the solution could be implemented using a single function.

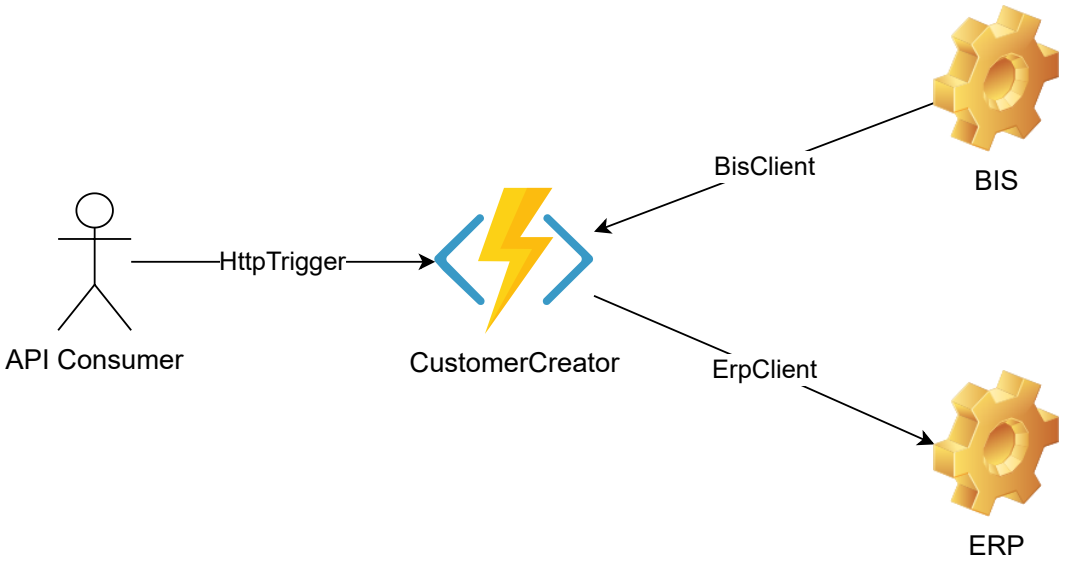


Figure 5.11: A diagram of the serverless implementation of Use Case 3

Moreover, this function was created using Java, the same programming language as the conventional implementation. This meant that considerable amounts of code from the conventional implementation could be reused in the serverless implementation, along with some of the dependencies. Nonetheless, parts of the reused code required refactoring to suit the demands of the Azure Functions platform.

For instance, the model classes are identical to the respective ones in the conventional implementation. Furthermore, the client classes, which are used for communicating with the BIS and the ERP, could also be reused in the serverless implementation without any significant changes. The only step needed was to add the required Spring Framework module dependencies into Maven, the dependency manager used in the implementation. In the conventional implementation, these dependencies are included in so-called Spring Boot starters, resulting in easier dependency management. However, this had to be done manually in the serverless implementation, as Spring Boot is not used. Moreover, this meant that the IoC mechanism or autoconfiguration provided by Spring Boot could not be utilised. Therefore, the clients had to be created and configured manually.

```java
@FunctionName("CustomerCreator")
public HttpResponseMessage run(
    @HttpTrigger(
      name = "req",
      methods = {HttpMethod.PUT},
      authLevel = AuthorizationLevel.FUNCTION,
      route = "v1/createCustomer")
      HttpRequestMessage<Optional<String>> request,
    final ExecutionContext context) {

  final String businessId = request.getQueryParameters().get("businessId");
  if (businessId == null) {
    return request.createResponseBuilder(HttpStatus.BAD_REQUEST).body("Please pass the business id on the query string").build();
  }
  try {
    Optional<ErpCustomer> response = createCustomerProfile(businessId);
    return response.isPresent()
        ? request.createResponseBuilder(HttpStatus.CREATED).body(response).build()
        : request.createResponseBuilder(HttpStatus.NOT_FOUND).build();
  }
  catch (BisException e) {
    context.getLogger().warning(String.format("Failed to retrieve business info from BIS, status %s, message: %s", e.getHttpStatusCode(), e.getMessage()));
    return request.createResponseBuilder(e.is4xx() ? HttpStatus.INTERNAL_SERVER_ERROR : HttpStatus.BAD_GATEWAY).body("Failed to create customer profile, could not retrieve business info from BIS").build();
  }
  catch (ErpException e) {
    if (e.getHttpStatusCode().equals(HttpStatus.CONFLICT)) {
      context.getLogger().info(String.format("Existing customer found in ERP with same business ID, no new profile created. businessId=%s", businessId));
      return request.createResponseBuilder(HttpStatus.CONFLICT).body("Existing customer profile already exists in ERP for given business id").build();
    }
    context.getLogger().warning(String.format("Failed to create new customer profile in ERP, status %s, message: %s", e.getHttpStatusCode(), e.getMessage()));
    return request.createResponseBuilder(e.is4xx() ? HttpStatus.INTERNAL_SERVER_ERROR : HttpStatus.BAD_GATEWAY).body("Failed to create customer profile, could not save to ERP").build();
  }
  catch (Exception e) {
    context.getLogger().severe(String.format("Exception occurred during the operation with businessId=%s, message: %s", businessId, e.getMessage()));
  }
  return request.createResponseBuilder(HttpStatus.INTERNAL_SERVER_ERROR).body("Failed to create new customer profile in ERP, something went wrong during the operation").build();
}
```

Listing 5.19: The main body of the CustomerCreator function

The serverless implementation differs primarily from the conventional solution

in the controller aspect. As the serverless implementation is a function, there is no separate controller class to handle the API endpoints. Instead, the function features an HTTP trigger that initiates the execution of the function. The main body of the function *CustomerCreator* is depicted in listing 5.19.

Listing 5.19 shows that the endpoint is configured using the Azure Functions-specific *@HttpTrigger* annotation. This annotation includes the HTTP request type, whether authorisation is required, and the route of the request. When the function is triggered, it first tries to read the given business ID of the request. An HTTP 400 response will be returned if one is not given. Otherwise, the function will call the *createCustomerProfile()* method with the given business ID as input. This method is the same as in the conventional implementation, which was previously depicted in 5.17.

The way the responses are returned is also similar to the conventional implementation. The logic within the try-block in listing 5.19 is nearly identical to the controller's logic in the conventional implementation, which was previously depicted in listing 5.16. The difference is in how the responses are created. The serverless implantation must use the *createResponseBuilder()* method provided by Azure Functions, as the platform requires this. Conversely, the conventional implementation uses *ResponseEntity* objects provided by Spring, which are created and returned.

The serverless implementation also performs the exception handling within the main body of the CustomerCreator function using basic try-catch blocks, as seen in listing 5.19. The exception handling was implemented in this manner because the function must return the response itself. Nevertheless, the logic within the catch-blocks is nearly identical to the handler methods of the conventional implementation. The custom exceptions, *BisException* and *ErpException*, are the same as in the conventional implementation, apart from some platform-specific differences.

As previously described, the HTTP trigger of the function requires authorisation for it to be initiated. The authorisation options are configured in the @HttpTrigger annotation in the *authLevel* parameter. As the parameter is set to *function*, a valid authorisation key must be attached to the HTTP request for the function to be triggered. The function will return an HTTP 401 response if the authorisation key is missing or invalid. Once deployed in the cloud, the authorisation keys may be retrieved from the Azure portal. The Azure Functions platform handles the keys' validation when the endpoint is called.

# 6. Comparisons

In this chapter, the conventional and serverless implementations covered in the previous chapter are compared against each other. This thesis aims to compare different aspects of the development process in addition to the implementations themselves. Moreover, some operational aspects are considered as well.

## 6.1 Use Case 1

The development process of the serverless implementation, which was predominantly done using Azure Logic Apps, differed noticeably from the development process of the Spring Boot Application. Instead of programming an application in a conventional sense using programming language and a framework, the developer uses a designer tool to create a linear workflow consisting of connectors, the building blocks of the business logic. The end results are two vastly different solutions which perform the same task.

Development time-wise, both implementations took a similar amount of time for the author to plan and implement. With the serverless solution in Azure Logic Apps, a considerable portion of time was spent understanding the designer itself, as it requires the developer to take a different approach and way of thinking to implement the solution. The designer works well when creating simple workflows utilising multiple managed connectors. However, some more complex operations, which a developer could express in code without issues, may require more effort to implement if they are not accustomed to the designer.

Furthermore, the designer features some aspects that feel unintuitive. For example, the evaluation of whether a scope succeeded or not, an operation previously seen in figure 5.4, requires the use of an expression function to return the status of the scope during runtime. This expression function is used in the subsequent conditional action, the settings of which are shown in figure 6.1. Moreover, the conditional action must also be separately configured to run regardless of the scope's status so that the result may be evaluated. A guide on how to accomplish this is described in the Logic Apps documentation [60]. However, this is unclear from the designer itself. The whole operation of adding an expression function to retrieve

the scope's status feels somewhat gimmicky and not something which is supported by design.

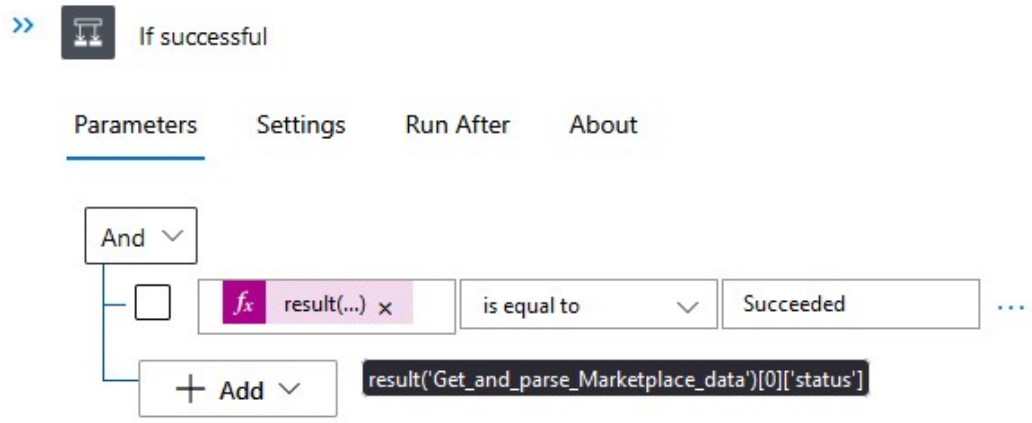


Figure 6.1: The evaluation of the scope's status in the serverless implementation

Nonetheless, once one becomes accustomed to Logic Apps and its designer, it may be used to implement certain operations quickly. This may be most apparent in the final stages of the serverless workflow, depicted previously in figure 5.7, where an email notification is sent to the billing department in case unknown companies are found. A built-in connector is initially used to parse and create an HTML table from the JSON containing the company licence usage data. Then, the table is sent as part of a message using the Outlook connector, which is connected to an email account within Office 365.

Compare this to the email-sending process in the conventional implementation. The email functionality is contained in a separate *EmailService* class, which uses *JavaMailSender* provided by Spring to send the email. *JavaMailSender*, in turn, sends the email through a custom SMTP server configured into the application configurations. Moreover, the text of the email is generated using a custom method in the EmailService class. The method takes a list of licence usage data as input and utilises it to generate a message.

Although Spring Boot's autoconfiguration can automatically read the SMTP-configuration data end setup the connection to the server, setting up the entire EmailService class still took considerably longer than adding two connectors to the serverless workflow in Logic Apps. Furthermore, the serverless implementation does not require a custom SMTP server to send the email, as this is accomplished using the connected email account instead. Logic Apps also contains a separate SMTP connector, which may be used when emails need to be sent using a custom server [61].

An important detail about the serverless implementation is that the connectors used in the workflow are predominantly built-in actions, with the Outlook connector being the only managed connector. One of the main advantages of Logic Apps is that the managed connectors allow developers to easily add powerful operations to workflows by abstracting away the inner workings. The Outlook connector, for instance, handles all the behind-the-scenes steps with the Outlook server to send the email. Moreover, the credentials used for the email account are also stored within Azure. As the service provider handles these steps, managed connectors leave more time for developers to design the business logic.

This is not to say that built-in connectors do not offer any benefits. As they are part of the Logic Apps runtime, it can be assumed that the service provider has ensured that they work correctly. This, in turn, does save some time during the testing phase. For instance, the author had to spend additional time writing unit tests for the conventional implementation to ensure that the code worked as intended. This also applied to the client classes responsible for sending the HTTP requests. On the other hand, in the serverless implementation, all HTTP requests are sent using the built-in HTTP action. As the built-in connectors can be assumed to work as intended, developers may instantly start testing that the actions are correctly configured and ensure that the implemented business logic functions as intended.

However, as the serverless implementation predominantly consists of built-in connectors, this result results in a considerable amount of custom logic being implemented using the designer. As this type of logic could easily be expressed in code, implementing it with the Logic Apps designer may not be efficient, considering that Logic Apps is better at orchestrating logic than executing it during runtime. Consequently, the generation of the data for the invoice charges is performed by Azure Functions.

The number of connectors used in the serverless implementation also means that the workflow is somewhat lengthy. Therefore, expanding it afterwards with more business logic may be challenging if the need arises. Moreover, the linear nature of workflows in Logic Apps may also be restrictive. Hence, it may be necessary to create new workflows to cover the new aspects of the required business logic.

The conventional implementation, in turn, required more effort to develop. The programming language Java combined with Spring can sometimes result in rather verbose code, requiring many lines to be written to perform relatively uncomplicated operations. Examples of this are the client classes responsible for communication with the Marketplace and ERP systems, as well as the email notification functionality. However, once the conventional solution was implemented and tested,

the result was a fully-fledged Spring Boot application. Consequently, the application may be more easily customised and expanded in the future compared to the Logic Apps workflow of the serverless solution.

From an operational aspect, the two implementations also differ. The conventional Spring Boot application does not feature any specific ties to Azure. Therefore, it may be deployed in practically any environment, in the cloud or on-premises. The operating costs will vary depending on the environment. In the case of on-premises, the operational costs of the application will be tied to the maintenance cost of the server. In the cloud, the vendor charges for reserved capacity.

The serverless implementation can, conversely, only be deployed on Azure Logic Apps, as it is developed for that platform. As is characteristic of serverless platforms, the operational costs of the workflow will be based on consumption. This, in turn, is highly dependent on the number of companies and licences retrieved from the Marketplace, in addition to the number of workflow executions. In production, the solution should run only once a month unless some error occurs.

As previously described in chapter 3.2, the consumption plan in Logic Apps meters each connector execution. The unit price for one execution varies depending on which group the connector is part of. The connector groups and their associated prices can be seen in table 6.1.

| Connector type | | Price in USD | |
|---|---|---|---|
| Built-in | Action<br>Trigger | 0.000025 | Per Execution |
| Managed | Standard<br>Enterprise | 0.000125<br>0.001 | Per Call[1] |

Table 6.1: The connector execution prices in Azure Logic Apps for the Europe-West region as of August 2022 [44][62]

The workflow of the serverless implementation contains in total of seventeen actions and one trigger. All of these belong to the built-in group, with the Outlook connector being the only exception. During a typical run, the nine built-in connectors will always be executed. In addition, for each company in the Marketplace, the loop will execute five built-in connectors if a corresponding customer profile is found. Assuming a profile is found for each company, the cost of running the workflow once can be calculated using the following formula, where $x$ represents the number of companies in the Marketplace:

[1]Certain managed connectors may require multiple API calls to perform some operation. In these connectors, each API call is billed separately.

$$\$0.000025 * (9 + 5x)$$

For example, if the Marketplace would contain 1 000 companies, then the total cost of running the workflow once would be:

$$\$0.000025 * (9 + 5 * 1000) = \$0.125225$$

However, if a company's corresponding customer profile is not found, then only three built-in connectors will execute within the loop instead. Moreover, this would result in the email notification being sent using the Outlook connector. This increases the base number of connector executions by one built-in and one managed standard connector. The Outlook connector requires one API call to send a single email [63]. Overall, this results in the following formula:

$$\$0.000025 * (10 + 5x + 3y) + \$0.000125$$

In the formula above, $x$ represents the number of known companies (with corresponding customer profiles in the ERP system) and $y$ the number of unknown companies. For example, assuming the Marketplace has 1 000 companies, out of which 20 are unknown, the total cost of running the workflow once would be:

$$\$0.000025 * (10 + 5 * 980 + 3 * 20) + \$0.000125 = \$0.124375$$

The total prices given in the calculations only include the costs associated with the Azure Logic Apps. The serverless implementation also uses Azure Functions to generate the data for the invoice charge. The costs associated with Azure Functions cannot be estimated precisely, as the number of execution units (GB-s) consumed varies depending on multiple factors. This includes the number of companies, the number of products, and the state of the memory working set of the function app. However, assuming the Marketplace contains 1 000 companies, the author estimates the execution cost to be less than 1 cent per workflow execution. Overall, this sum is negligible in comparison to the costs associated with Logic Apps.

## 6.2 Use Case 2

The serverless solution was developed using Azure Functions in combination with the Durable Functions extension. Moreover, Azure Cosmos DB was used for persistent storage. Compared to the previous use case, the development process of the

serverless solution overall felt more familiar to the author, as the business logic was implemented using code. This resulted in many similarities between the serverless and the conventional implementations, with some lines of code being more or less identical. However, there are still significant differences between the two implementations in multiple areas.

The conventional and the serverless implementations took approximately the same time to implement, though the serverless took slightly longer. This was primarily due to two reasons. Firstly, the author had less experience in C# development compared to Java. Although the programming languages are similar in many ways, C# still features its own .NET-related dependencies, which differ from those used in Java. Secondly, the author was not previously familiar with Azure Functions or the Durable Functions extension. Therefore, understanding the details of the platform and how the business logic should be structured required some familiarisation.

Moreover, it takes time to comprehend the bindings and how they should be utilised. The Azure Functions platform features multiple bindings for the same service, resulting in various ways to implement the same logic. For instance, Cosmos DB features at least seven alternative ways to retrieve data from the service using different bindings [64]. However, once one begins to understand the bindings, they allow the developer to perform a variety of operations within Azure. Microsoft also has on their *docs.microsoft.com* website featuring comprehensive documentation of all the available bindings, including examples in different programming languages supported by Azure Functions. Similar documentation is also available for the Durable Functions extension.

In the serverless implementation, the bindings are primarily used for exchanging data with Cosmos DB. This is comparable to how Spring Data JPA repositories are used in the conventional implementation. Both are straightforward to use and abstract much of the complexity for the developer, allowing database operations to be conducted with ease. Both also feature their distinct objects for representing data. In the conventional implementation, these are in the form of JPA entities. Conversely, the serverless implementation uses DTOs, as the Core API of Cosmos DB stores data in the form of JSON documents.

Both implementations orchestrate their business logic in a structured manner. In the conventional application, the *runBilling()* method within the *RunService* class is responsible for the orchestration, whilst the serverless implementation features a separate orchestrator function. However, due to the nature of the serverless platform, the business logic is more clearly structured in the serverless implementation. The solution consists of six activity functions, each with its own assigned task in the

workflow. Each activity function's task can be directly interpreted from their names. For instance, the *ProjectRetriever* function is responsible for retrieving all projects. In the conventional implementation, one has to delve deeper into the classes and their respective methods to understand how the logic is structured.

A considerable difference between the two implementations is that the serverless version features parallel processing of the projects. This was included because the Durable Functions extension made adding such functionality straightforward. The developer only needs to instantiate new function instances in the orchestrator. Meanwhile, the platform handles the creation of the new function instances and their resource allocation in the cloud. Moreover, the platform makes the orchestrator function wait as long as needed until all the function instances have finished executing before continuing. Overall, this results in performance benefits when processing a large number of projects.

Implementing multi-threading in the conventional implementation is by no means impossible. However, this would require more effort compared to the serverless implementation, as the developers must implement such functionality themselves rather than relying on the platform to do it for them. The time required varies depending on the developers' experience. However, by manually implementing parallelisation, the developers gain more control regarding the threads, which may allow for more efficient processing.

Nevertheless, all the parallelisation in the conventional implementation would still happen within the context of the Spring Boot application. Thus, the performance would ultimately be tied to the hardware the application is deployed on. Moreover, the database used may impose performance restrictions if it is unable to keep up with the requests.

The serverless implementation does not suffer from similar restrictions as the cloud offers virtually unlimited access to computational resources. Therefore, an individual function app can have hundreds of functions executing simultaneously. The serverless mode of Cosmos DB also does not have any throughput limitations, allowing a substantial amount of database operations every second. However, as previously mentioned in chapter 3.3, the performance in the serverless mode does not feature performance guarantees.

Another helpful feature provided by the Durable Functions extension is the ability to retrieve the status of the orchestration through the built-in API. This can be done both during and after the orchestrator has finished executing, as statuses are retained in the Azure Storage account even after the execution has ended. The conventional implementation only features logging to report on the status of the

execution. Of course, as with multi-threading, the developers could implement customised status reporting themselves. However, this requires more time and effort. The added benefit would be that the functionality could be better customised to fit the needs of stakeholders.

Both solutions utilise databases to store billing data during the billing run. The format and methods differ even if the data stored in both solutions are identical. The conventional implementation uses Spring Data JPA to store data in a relational database, while the serverless implementation uses Cosmos DB, a NoSQL database. The type of database used is in itself not relevant to the functionality of the implementations. However, a noteworthy difference is that the serverless version is tied directly to a specific type of database, whilst Spring Data JPA acts as an abstraction in the conventional implementation. This abstraction means it is possible to alter the type of database used in the conventional implementation without the need to modify the application code. Meanwhile, the serverless implementation is directly dependent on Cosmos DB, as the code contains bindings and other elements specific to the service.

Spring Data JPA and Cosmos DB both feature learning curves. The former is widely used within Java development. Consequently, the author already had some previous knowledge of using it. On the other hand, Cosmos DB is a more recent proprietary database service. Thus, it features some aspects many developers may not be familiar with, such as containers and partition keys. Partition keys, in particular, are essential to the service, as they impact how data is stored and retrieved from containers. Moreover, partition keys must be included when performing some database procedures.

In the conventional implementation, all database operations are managed by Spring Data JPA. Basic CRUD operations are supported by default. For more specific operations, such as finding a billable associated with a particular project, the developers only need to create a method in the repository interface that follows the given naming convention. Spring Data JPA can then derive the database query needed to retrieve the data from the method names. Moreover, developers may generate these methods using tools such as JPA Buddy. Cosmos DB also supports basic CRUD operations, although some require the partition key to be included in the procedure. However, more specific operations may require the database query to be manually written in addition to logic to iterate on the data, as seen in listing 5.11.

Common for both implementations is that local development works well. Although Azure Functions and Cosmos DB are both cloud services, each features

tools that support local development. For Azure Functions, a local version of the runtime is included as part of Azure Functions Core Tools, which allows developers to test their functions on their personal computers. The Durable Functions extension requires a connection to an Azure Storage account, which can also be emulated locally using Azurite. Likewise, Cosmos DB also features an emulator for local development.

From an operational aspect, the two implementations differ in many ways. The conventional implementation must be deployed on a dedicated set of computational resources. However, as the Spring Boot application is not tied to Azure, it may be deployed in practically any environment, whether in the cloud or on-premises. Similarly, the required database can be located in any environment. The database type is irrelevant as long as Spring Data JPA supports it.

Conversely, the serverless implementation can only be deployed within Azure. In the consumption-based serverless plan, costs only accumulate when computational resources are consumed. Similarly to the previous use case, this depends on the number of times the orchestration will be executed and the amount of data processed. Furthermore, additional costs will be associated with consumed storage in the Azure Storage account and Cosmos DB. However, as the data generated is only in the scale of megabytes and is only stored for short periods, the costs are negligible compared to the execution costs. Thus, they will be overlooked.

In Azure Functions, the number of consumed resources is measured in gigabyte-seconds (GB-s). To measure the GB-s consumed by the serverless implementation, the author deployed the solution on Azure's Europe-West region and performed test runs. During each test run, the billing engine retrieved and processed 1000 projects, with each project containing 10 resources. The resources were all created and deleted during the billing period. This meant that the billing engine processed 20 resource events for each project. The results of the test runs are displayed in tables 6.2, and 6.3. Table 6.2 contains the execution metrics for Azure Functions, whilst table 6.3 includes metrics for Cosmos DB. Furthermore, the former table also contains the execution time of the test runs.

The cost calculations are based on the prices for the West-Europe region as of August 2022. As seen from the tables, the execution cost of a single run is approximately 0.10 USD on average. Only a marginal portion of this cost comes from the computation process in Azure Function. The serverless implementation consumed around 150 GB-s per billing run. As the cost of one GB-s is only 0.000016 USD [65], the cost mounted up to less than a quarter of a cent per billing run.

Nearly all of the costs associated with the test runs resulted from the database

| Run # | Execution time (in seconds) | MB-ms (in millions) | GB-s | Cost in Azure Functions (in USD) | Function Execution Count |
|---|---|---|---|---|---|
| Run 1 | 89 | 150.81 | 147.275 | $0.00236 | 3120 |
| Run 2 | 139 | 149.20 | 145.703 | $0.00233 | 3140 |
| Run 3 | 103 | 153.18 | 149.590 | $0.00239 | 3090 |
| Run 4 | 101 | 151.50 | 147.949 | $0.00237 | 3090 |
| Run 5 | 121 | 157.10 | 153.418 | $0.00245 | 3040 |
| Average | 110.6 | 152.36 | 148.787 | $0.00238 | 3096 |

Table 6.2: The test run metrics from Azure Functions

| Run # | Peak Server-Side Latency (in milliseconds) | RUs consumed (in thousands) | Cost in Cosmos DB (in USD) |
|---|---|---|---|
| Run 1 | 12.91 | 339.57 | $0.1036 |
| Run 2 | 7.48 | 339.66 | $0.1036 |
| Run 3 | 34.70 | 339.84 | $0.1037 |
| Run 4 | 38.57 | 339.74 | $0.1036 |
| Run 5 | 35.16 | 339.87 | $0.1037 |
| Average | 25.76 | 339.74 | $0.1036 |

Table 6.3: The test run metrics from Azure Cosmos DB

operations in Cosmos DB, with each run consistently consuming almost 340 000 request units (RUs). As the cost of 1 million RUs is 0.305 USD in the serverless mode of Cosmos DB [58], the total ends up being slightly over 10 cents per test run.

There is a slight variation in the number of resources consumed, more so on the computational side in Azure Functions than in Cosmos DB. The differences are presumably due to the FaaS platform having to provision the computational resources and perform the computations in a shared cloud environment, causing some variance. Moreover, there is also some variation in the number of function executions. This is due to the orchestrator function executing several times during each run. The Durable Functions extension stores and restores the state of the orchestrator function repeatedly as activity functions are executed. The number of times this is done is not constant.

Furthermore, there is some variance in the peak server-side latency in Cosmos DB. As mentioned in Chapter 3.3, Azure does not offer similar performance guarantees for the serverless mode of Cosmos DB as for the provisioned throughput mode [53]. Therefore, at the beginning of the run, the peak latency may be slightly higher. For comparison, in provisioned throughput mode, the latency is by the SLA

set be less than ten milliseconds [66]. Nonetheless, after the initial peak, the latency drops to less than five milliseconds for the rest of the run, as seen in figure 6.2.

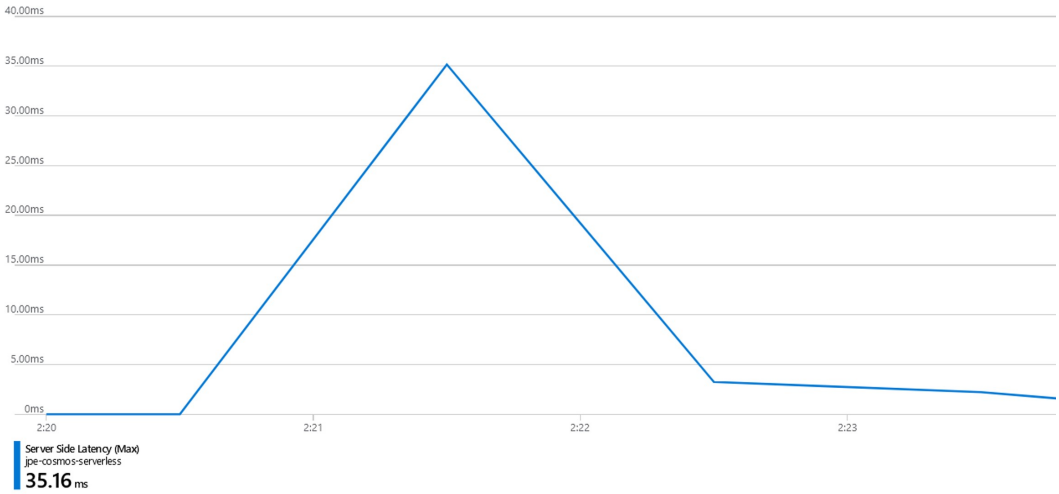


Figure 6.2: The server-side latency in Cosmos DB of Test Run # 5

The total execution time of the test runs in the serverless implementation was less than two minutes on average. The orchestrator function distributes the multiple activity functions to reduce the overall execution time. To see if this parallelisation brought any performance benefits, the author ran the conventional solution locally using the same test data of 1000 projects. The results of these test runs are shown in table 6.4.

| Run # | Execution Time (in seconds) |
|---|---|
| Run 1 | 249.732 |
| Run 2 | 305.926 |
| Run 3 | 276.870 |
| Run 4 | 270.704 |
| Run 5 | 241.945 |
| Average | 269.354 |

Table 6.4: The execution times of the test runs for the conventional implementation

As seen from table 6.4, the execution time of the conventional implementation was, on average, almost 270 seconds. Although a production-grade server may provide a more capable execution environment, this is somewhat offset by the application having a local database available for the test runs, which resulted in no additional network delays.

## 6.3 Use Case 3

The third use case differs from the two previous ones in that both the conventional and the serverless implementations are written in the same programming language, Java. This shared language resulted in many similarities between the two solutions, with some methods and classes being identical. Nevertheless, there are distinctions between the two solutions, which result in various impacts.

The business logic of the relatively simplistic third use case meant that the solutions did not take a long time to implement. The serverless version, in particular, was implemented in a concise time frame. Primarily, this was because code written for the conventional implementation could be refactored to be used in the serverless version. The possibility to use the same dependencies as the conventional version meant that the *client* classes, which are used for communication with the BIS and ERP system, could be reused by only making small changes. Moreover, the rest of the business logic could easily be adapted to fit the requirements of the Azure Functions platform.

Another reason for the brief development time was the overall simplicity of the serverless implementation, as Azure Functions manages the inner workings of triggering and securing the endpoint. The developer merely needs to set the appropriate values in the *@HttpTrigger* annotation, after which they can start implementing the business logic. The same applies partly to Spring Boot as well, as the opinionated autoconfiguration handles much of the complexity. This can well be seen in the controller of the conventional implementation, where similarly to the serverless version, only some annotations need to be added to inform the Spring runtime which classes and methods should handle HTTP requests.

However, even some manual configuration may be required even if Spring Boot is used. For instance, in the conventional implementation, the default autoconfiguration for Spring Security was unsuitable since it, among other things, resulted in the application featuring its own login page. This was not intended and had to be disabled along with some other features, whilst others had to be modified. Overall, this is not an issue, but it does result in more work. Moreover, if a developer makes a mistake while creating the security configurations, it may have serious consequences. However, manually creating some or all configurations gives the developers more control, allowing the solution to be customised in a way that cannot be done using a platform such as Azure Functions.

The implementations also differ regarding how their functionality may be expanded in the future. As the conventional implementation is a fully-fledged Spring

Boot application, the API could easily be expanded by adding new methods to the existing controller class. The new endpoints could take full advantage of the existing exception handlers in the controller and the functionality offered by the service class. Moreover, if the need arises to create entirely new controllers, the existing exception handlers could be moved to a separate *ControllerAdvice* class. Thus, the same exception handlers could handle the exceptions for the entire Spring Boot application. Overall, this would result in less duplicated code that would be easier to manage.

On the other hand, the serverless implementation cannot similarly be expanded with new functionality. Adding a new endpoint to the API would require creating an entirely new function. Nevertheless, the new function could be part of the same function app, allowing it to utilise some classes and methods of already existing functions. However, sharing functionality in the same manner as in Spring Boot applications is impossible. For example, each function requires its own exception handling, which could result in some duplicated code. However, some exception-handling logic could theoretically be shared by placing it in a separate class.

Performance-wise, there are also differences between the two implementations. As described in chapter 2, serverless is affected by a phenomenon known as cold starts, caused when the cloud provider has to allocate new computational resources from zero. How impactful this is varies depending on the use case of the application. In the case of APIs, cold starts can be particularly noticeable, as it may take longer than expected for the application to serve a response.

To test the occurrence of the cold start problem in the serverless implementation, the author deployed the solution to Azure's Europe-West region and ran some benchmarks. When calling the endpoint after the application had been inactive, an additional delay of around two to four seconds occurred on average. The application responded within a few milliseconds for subsequent requests made after the cold start delay. The shortness of the delay is likely due to the application being lightweight. Thus, the application can instantly execute once the cloud provider has provisioned the container.

A delay of a few seconds is likely not to be a dealbreaker for APIs such as the one in Use Case 3. As the API is intended only to be used internally, a slight reduction in response time is unlikely to have a considerable impact. Moreover, if the API is subsequently called, responses following the initial call will be delivered without additional delay.

The conventional implementation does not feature similar delays as the application is constantly running. Consequently, the solution requires dedicated computa-

tional capacity in the cloud or on-premises. Reserved capacity in the cloud means that the operational costs will be the same regardless of how often the API is consumed. In the case of on-premises, the operational costs will be tied to the server it is deployed on.

The operational costs of the serverless implementation vary depending on usage. In Azure Functions, consumed resources are measured in units of gigabyte-seconds (GB-s). Tests were performed to measure the number of GB-s consumed by the serverless implementation. The author performed tests while the solution was deployed on Azure's Europe-West region. During each test run, the application performed one regular operation: retrieving company details from the BIS and inserting the customer profile into the ERP system.

The execution costs, depicted in tables 6.5 and 6.6, varied depending on the present memory working set of the function app. As seen from the tables, the execution cost was over three times higher when the API was called when the function app was idle. Therefore, the cloud vendor has to start the application and allocate resources, which raises execution costs.

| Test # | GB-s | Cost in USD (0.000016/GB-s) |
|---|---|---|
| Test 1 | 2.207 | $0.00003531 |
| Test 2 | 2.256 | $0.00003609 |
| Test 3 | 2.539 | $0.00004063 |
| Test 4 | 2.666 | $0.00004266 |
| Test 5 | 2.178 | $0.00003484 |
| Average | 2.369 | $0.00003791 |

Table 6.5: The execution cost when starting from idle

| Test # | GB-s | Cost in USD (0.000016/GB-s) |
|---|---|---|
| Test 1 | 0.691 | $0.00001106 |
| Test 2 | 0.592 | $0.00000946 |
| Test 3 | 0.758 | $0.00001214 |
| Test 4 | 0.727 | $0.00001162 |
| Test 5 | 0.649 | $0.00001038 |
| Average | 0.683 | $0.00001093 |

Table 6.6: The execution cost of the function with existing memory working set

In addition to the GB-s consumed by the function executions, tables 6.5 and 6.6

also contain the price in USD. The costs are calculated based on the prices for the Europe-West region as of August 2022. As seen from the tables, the cost of a single execution is marginal, regardless of the state of the memory working set.

Nevertheless, even small costs can mount up to more considerable sums if the function is repeatedly executed. This can be relevant especially for APIs, as they can be consumed in large numbers within a short period. However, considering the use case of the implementation, which is an internal API, it is unlikely that the number of function executions would surpass a significant amount. Even then, in a worst-case scenario, the cost of one million function executions would amount to approximately 38 USD when using the average price displayed in table 6.5.

# 7. Discussion

This chapter broadly discusses different advantages and drawbacks noted during the implementation and comparison phases. The future implications of each alternative are also touched upon.

## 7.1 Noted Benefits of Serverless

The serverless implementations of the use cases demonstrate that serverless platforms offer some unique benefits. The level of abstraction decreases complexity, allowing developers to focus more on implementing the business logic itself. This was most apparent in the serverless implementation of Use Case 1, which used Azure Logic Apps as the development platform. The visual designer allows for the creation of linear workflows consisting of connectors acting as building blocks. As such, it is possible to implement business logic without the need to write any code.

However, the platform works best for implementing use cases that can benefit from the existing set of managed connectors available in the designer. Managed connectors can trigger the execution of a workflow in response to something more intricate or perform some powerful procedure, both within and outside of Azure. For instance, managed connectors could be used to transfer a file using SFTP when uploaded to a particular Azure Storage account or notify a developer in Slack whenever a work item is assigned to them in Azure DevOps.

The built-in connectors, used primarily for more basic operations, can still be valuable. However, when a use case mandates custom logic that results in a lengthy workflow with numerous connectors, the question arises whether using code would be more efficient. Creating comprehensive workflows with conditional loops and scopes requires a more profound understanding of the designer, which many developers may not be familiar with beforehand. Moreover, with the billing model of Logic Apps, repeatedly executing a workflow consisting of numerous connectors may cause high operational costs.

The serverless implementation of Use Case 1 is an excellent example of this. The workflow only utilises one managed connector, leading to the business logic largely being assembled using the built-in connectors. Although the solution is

only intended to be executed monthly, the loop it contains leads to higher operating costs compared to other alternative serverless platforms, such as Azure Functions. Therefore, such platforms should be considered in future similar use cases.

The serverless implementations of Use Cases 2 and 3 utilise Azure Functions as their primary development platform. Compared to Logic Apps, the FaaS platform feels more familiar to most developers, as the logic is implemented using code. Nevertheless, even Azure Functions require some familiarisation. This primarily applies to the various bindings used to exchange data in and out of functions. Moreover, the Durable Functions extension introduces new features related to function orchestration.

However, once one gains an understanding of these features, Azure Functions may be used to perform various types of computations. The serverless implementation in Use Case 2 demonstrates that the FaaS platform combined with the Durable Functions extension can even be used to perform tasks mandating more complex orchestration. Furthermore, the extension made it effortless to introduce parallelisation to the processing, which improved the overall performance compared to the conventional implementation.

Use Case 2 also demonstrated that serverless is well fit to handle bursty applications. During a billing run, the solution processes a large number of projects at once. However, as this is performed infrequently, reserving dedicated computational capacity may not be cost-effective. The consumption-based service model of serverless platforms means that the user only pays for consumed computational resources. This means no additional costs will be associated with the serverless implementation while it is inactive.

Moreover, when a serverless application is executed, the cloud provider allocates and scales computational resources as needed. In the case of Use Case 2, the implementation could also take advantage of the serverless mode of Cosmos DB. The serverless mode is well suited for situations where a high amount of throughput is occasionally needed, such as in bursty applications.

## 7.2 Noted Disadvantages of Serverless

Although serverless offers many advantages, there are drawbacks as well. One of the most apparent is the cold start phenomenon, which introduces a slight delay whenever the execution of a serverless application is initiated from idle. How impactful this delay is varies depending on the application's use case. In Use Case 2, the delay does not impact the overall performance. However, in Use Case 3,

the cold start introduced an additional delay of around two to three seconds to the response time.

For an internal API such as the one in Use Case 3, such a delay should not cause considerable disturbances. Nonetheless, cold starts may rule out use cases that require almost instantaneous responses, such as real-time applications. Furthermore, in the case of APIs meant for external consumers, whether such a delay is acceptable or not, is up for debate.

The delay caused by cold starts may also be amplified in micro-service environments. In such environments, one operation may require multiple distinct micro-services to be called consecutively. If each service is deployed separately on a serverless platform, then every service call would introduce a new cold start delay in case the service is idle. Suddenly, only a two-second delay in one service becomes a six-second delay for an entire operation involving three separate services.

From a non-functional perspective, a significant drawback with serverless platforms is that they introduce hard dependencies to the platforms themselves and the cloud vendor more broadly. This can cause concerns, as applications would then be relying on the services of a specific cloud provider, thus preventing them from being deployed elsewhere. This type of vendor lock-in is immediately apparent in the serverless implementation of Use Case 1, which was created using a designer tool specific to Azure Logic Apps. Consequently, the implemented workflow can only be deployed on Logic Apps. Deploying an equivalent solution elsewhere would require a complete reimplementation of the business logic.

Similarly, the serverless implementations of Use Cases 2 and 3 can only be deployed on Azure Functions. Although Java code is universal, the code of the serverless implementations contains dependencies specific to the FaaS platform, such as input and output bindings. This is most apparent in the serverless implementation of Use Case 2, which utilises the Cosmos DB bindings for communication with the service. Moreover, the business logic is orchestrated using the Durable Functions extension. Consequently, the solution also depends on an Azure Storage account in addition to the Azure Functions runtime itself.

Refactoring the logic of FaaS applications to fit other platforms, serverless or not, is not impossible. However, depending on the type of dependencies used in the application, doing so could require substantial effort. One way to mitigate this problem, suggested by a study [13], is to package the business logic into custom libraries. The logic specific to the FaaS platform, such as the bindings in the case of Azure Functions, could then be implemented around the custom library. However, as the study pointed out, this does not work in all circumstances.

## 7.3 What About the Conventional Implementations?

The conventional implementations for all three use cases are unaffected by such vendor-lock-related issues, as they are not restricted to a specific platform. The Spring Boot applications can be deployed in practically any environment, in the cloud or on-premises. Furthermore, Spring Boot introduces abstractions around many concrete dependencies. For instance, in Use Case 2, all database operations are handled by Spring Data JPA. This type of abstraction makes it possible to use nearly any type of database along with the application, as long as Spring supports it. Conversely, the serverless implementation has a hardcoded dependency on Cosmos DB. Therefore, the database cannot similarly be changed by merely modifying the application configurations, but major code refactoring is needed instead.

Another benefit of the conventional implementations is that they allow for more customisability. By simplifying the development process, serverless platforms also reduce the amount of control. When creating a Spring Boot application from the ground up, all features of the application can be customised to fit virtually any business need. The drawback is that this may require more effort than serverless platforms, where the cloud vendor manages at least some of the complexity. The parallelisation performed by the serverless implementation of Use Case 2 is an excellent example of this. However, the effort required to implement any feature in any application ultimately depends on the skills of the available developers.

Spring has the benefit of being already well established within the Java development scene. With the first milestone version of Spring released in 2004 [26], it is today the most popular framework for Java backend applications [23]. Therefore, the number of developers familiar with the technology should be higher than emerging technologies, such as serverless. For instance, Azure Functions was launched in 2016, the latter half of the previous decade. However, Azure Functions and other FaaS platforms have the advantage of supporting multiple programming languages, making the technology more approachable. Although multiple languages are supported, some features of the FaaS platform may be lacking. For example, this year, the Java version of Azure Functions finally gained support for the Durable Functions extension. Regardless, due to the Cosmos DB binding still lacking support, the serverless implementation of Use Case 2 had to be programmed using the C# language.

## 7.4 The Cloud – Less Upkeep by Renouncing Control

The conventional implementations, although more customisable, also require more upkeep. The dependencies used by Spring Boot applications should be kept up to date. Unless updated, the dependencies may later result in security vulnerabilities. For zero-day vulnerabilities, it is vital that mitigating measures are instantly taken. A recent example of this is the Log4Shell vulnerability, discovered last year [67].

In serverless platforms, the level of abstraction means that the cloud provider handles much of this upkeep. This is best seen in Logic Apps, where everything, including the designer, is managed by Microsoft. In Azure Functions, Microsoft is similarly in charge of maintaining the Functions runtime and the different bindings. However, if a function includes additional dependencies not part of the platform, these need to be updated manually. Nevertheless, the bindings and other features included by the FaaS platform may remove the need to add some dependencies in the first place, such as loggers.

Moreover, in the conventional implementations, another significant possible source of upkeep is the maintenance of the different layers of the computing stack, which are depicted in figure 7.1. The colours illustrate the different aspects of the stack. Which areas are managed by whom varies depending on where the application has been deployed. The green colour represents the applications themselves and any associated data. When deploying on-premises, each of these layers requires some upkeep. In IaaS, the cloud vendor manages the infrastructure portion of the stack, coloured in blue. Although IaaS delegates a considerable portion of the responsibility to the cloud provider, the yellow-coloured portion of the stack, the runtime environment, remains to be maintained. The runtime environment includes various pieces of software, stretching from the operating system itself to application containers, database engines and language runtimes. As maintaining the runtime environment can be tedious, cloud vendors have introduced different PaaS offerings where they also manage this layer. Serverless can be seen as an extension to PaaS, where the level of abstraction is taken one step further, and the user only pays for the consumed computational resources.

The serverless implementations do not feature similar deployment flexibility. Instead, they can only be deployed within Azure's cloud services. Although the cloud provides many benefits, some characteristics may make it unsuitable in certain circumstances. One of these is that users share the cloud with other customers. Shared computational resources are one of the key factors that enable cloud providers to offer serverless platforms in the first place. However, this also places

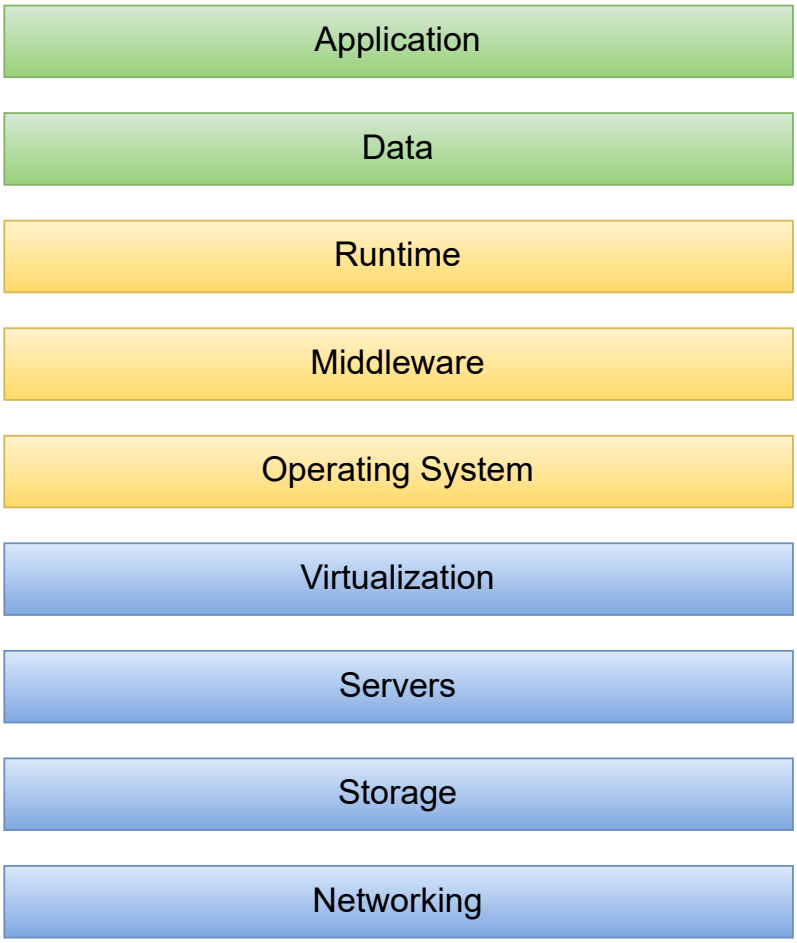


Figure 7.1: The different layers of the computing stack

a great deal of responsibility on the cloud provider to keep the data of different customers separate [17][68].

Even if the data is kept separate, the location of the data might still be an issue. For instance, as of August 2022, there is currently no Azure data centre located in Finland. Therefore, Finnish customers have no choice but to transfer their data abroad to use Azure's cloud services. Although data transfer within the European Union is generally acceptable, the cloud may not be a viable option if some data mandates domestic processing or retention. Moreover, even if a cloud provider's data centre is located domestically, there may still be a risk that sensitive data is accessible by foreign authorities [69].

The sharing of computational resources also means cloud providers cannot provide dedicated IP addresses for all services. This is particularly relevant for many serverless platforms, as the computational resources will be assigned from a shared resource pool upon execution. Not having a dedicated IP address is not always an issue. However, there are use cases where it is mandated.

However, the dependency aspect may be the biggest hindrance to serverless adoption. In the case of business-critical applications, it may not be feasible to create them using technologies tied to a specific cloud vendor. As previously discussed, serverless, in particular, suffers from the vendor lock-in problem. If the need arises to deploy the serverless application elsewhere, it may not be possible without considerable effort. Therefore, deciding whether or not to utilise serverless technologies is always a business decision, and the drawbacks of the technology should always be considered before committing to it.

# 8. Conclusion

Serverless platforms aim to simplify application development, deployment, and operation. By abstracting away the complex inner workings, the platforms allow developers to focus primarily on implementing the application's business logic. The purpose of this thesis was to investigate how serverless platforms may be utilised in a number of use cases. Moreover, the thesis compared the serverless implementations against conventional counterparts, which were implemented as Spring Boot applications.

The serverless implementations benefitted from many of the characteristics provided by the serverless platforms. Once the developer has familiarised themselves with the different elements of the platform, they can be used to add more complex features to the applications effortlessly. In addition, other characteristics, such as consumption-based billing models and resource scaling, made the solutions particularly suitable for bursty applications. The conventional implementations were more intricate than their serverless counterparts, particularly regarding some features. However, as the end results were proper Spring Boot applications meant that they could better be expanded with new functionality in the future, should the need arise.

The serverless platforms did not feature any limitations from a functional aspect concerning the investigated use cases. The cold start phenomenon may be a limiting factor in scenarios where short response times are mandated, such as real-time applications. Furthermore, if multiple serverless applications are consecutively called from idle, the delay caused by cold starts may accumulate.

As the functional limitations are few, the main deciding factors concerning the use of serverless platforms are the non-functional aspects. Although serverless abstracts much of the complexity, it consequently ties the application to the cloud vendor's platform, causing vendor lock-in. Moreover, this problem is amplified due to the serverless platforms integrating well with other services within the vendor's ecosystem. Moving serverless applications elsewhere is not straightforward and may even require a complete reimplementation of the business logic. The conventional implementations do not have similar restrictions and may be deployed practically in any environment.

Therefore, the decision of whether to utilise serverless platforms should be taken with care. The level of abstraction brought by the platform can help reduce the development time and operational burden. However, firmly tying an application to a specific cloud provider might pose a business risk.

## 8.1 Limitations

The findings presented in this thesis are based on the implementations of the example use cases described in chapter 4. Although most findings can be applied more broadly, they are unable to cover every scenario. Therefore, it is to be expected that other use cases exist containing aspects not considered in this thesis.

The implementations presented in chapter 5 were also created using a specific technology set. Notably, the serverless implementations only utilised cloud services provided by one cloud vendor. Consequently, even if serverless platforms generally feature similar characteristics, the findings may not be applicable to every platform. Moreover, the comparisons may yield different results should the technologies used for either implementation change.

## 8.2 Future Work

One of the noted benefits of the conventional implementations was that the technologies utilised are well established. Spring is the most used framework for Java backend development and, as such, is widely known among developers. Therefore, businesses and other organisations should find it easier to hire developers with previous experience of such technologies in the labour market.

As serverless platforms are still relatively new, it is unclear how well known they are in the developer community. How well established a technology is may be a contributing factor to whether organisations decide to invest in it or not. Therefore, the level of knowledge and awareness of serverless technologies among developers could be an area worth exploring.

# Summary in Swedish – Svensk sammanfattning

## Fördelar och nackdelar med serverslösa tillämpningar

### Inledning och bakgrund

I samband med utvecklingen av molntjänster och virtualiseringstekniker har nya sätt att distribuera och utveckla applikationer dykt upp som faller under begreppet serverlös databehandling. Trots att begreppet *serverlös* får det att låta som några servrar inte längre skulle alls behövas, så stämmer detta inte i det aktuella sammanhanget. Snarare betyder begreppet att servrarna och annan underliggande infrastruktur har abstraherats bort från användarna. Detta gör det möjligt för utvecklare att driftsätta och underhålla programvara i molnet utan att behöva tänka på hur det egentligen körs. Molnleverantören tar i stället hand om de operativa aspekterna genom att dynamiskt allokera datorresurser vid behov när programmet ska exekveras [1][2].

Serverlösa teknologier påminner på vissa sätt om olika Platform-as-a-Service-tjänster (PaaS), där molnleverantören också sköter de operativa aspekterna av den underliggande infrastrukturen. Skillnaden är dock att serverlösa teknologier tar detta ett steg vidare genom att ha utvecklarna att implementera affärslogik med hjälp av abstraheringar på högre nivå. Affärslogiken kan köras direkt i molnet utan behov att inpacka det som en del av en större applikation. Exekveringen av affärslogiken utlöses som svar på händelser som utvecklaren har definierat [1][3].

Det finns olika sorters serverlösa plattformar, som dessa program kan utvecklas för. Den allmännaste modellen kallas för Function-as-a-Service (FaaS), där utvecklare skapar funktioner bestående av kod. Dessa funktioner kan sedan självständigt köras i molnet. Flera funktioner kan även grupperas ihop för att skapa ett program, där varje enskild funktion utför en del av ett större arbetsflöde. Oftast stöder de olika FaaS-tjänsterna flera programmeringsspråk, vilket gör det möjligt för organisationer och utvecklare att välja mellan ett eller flera språk som är bäst anpassade för deras behov [2][3]. Molnleverantörer har även kommit ut med icke-FaaS-baserade serverlösa plattformar där affärslogik kan implementeras utan någon kod, som Azure Logic Apps från Microsoft [8].

Serverlösa plattformar har också en faktureringsmodell som skiljer sig från traditionella molntjänster, som vanligtvis faktureras baserat på reserverad kapacitet. Inom serverlösa tjänster betalar användaren däremot enbart för de resurser som förbrukats under exekveringen av ett program, vilket mäts av molnleverantören. Denna

typ av faktureringsmodell är möjlig eftersom inga datorresurser är särskilt reserverade för enskilda program, utan molnleverantören tilldelar dessa dynamiskt när de behövs.

Denna bruksbaserade faktureringsmodell kan göra serverlösa tjänster enormt förmånliga. Kunden undviker inte bara att behöva ställa upp någon infrastruktur för att sätta programmet i drift, utan behöver dessutom enbart betala för de resurser som förbrukas när programmet körs. Den här typen av faktureringsmodell är dock inte alltid billigare jämfört med reserverad kapacitet. Till exempel i situationer där resursförbrukningen av ett program är mycket hög under en längre period kan driftskostnaderna faktiskt vara dyrare.

Serverlösa plattformar har också vissa funktionella nackdelar. En av de mest märkbara är ett problem som kallas för kallstart. Eftersom serverlösa program inte har några särskilt reserverade datorresurser i molnet, måste molnleverantören varje gång allokera dessa från noll när programmet ska exekveras efter en stilleståndsperiod. Detta kan leda till en fördröjd start och försämrad prestanda i början av programmets exekvering [3].

En annan nackdel med serverlösa plattformar är att program som har utvecklas för en särskild plattform inte enkelt kan överföras till en annan. Detta beror huvudsakligen på att alla molnleverantörer har egna specifikationer, verktyg och krav för sina serverlösa plattformar. Problemet förvärras ytterligare av att serverlösa plattformar fungerar väl tillsammans med molnleverantörernas andra tjänster. Detta leder ofta till att även dessa tjänster utnyttjas av serverlösa program. Att ersätta dessa tjänster med alternativ kan vara ansträngande och leda till höga kostnader [13].

## Mål

Målet med den här avhandlingen är att jämföra hur programvaruutveckling med serverlösa teknologier skiljer sig från traditionell programvaruutveckling. För att göra dessa jämförelser har en serverlös implementering och en konventionell implementering utvecklats för tre olika exempel användningsfall. Användningsfallen är utformade för att förevisa en uppsättning av situationer som IT-organisationer kan ställas inför. Implementeringarna och deras utvecklingsprocesser jämförs sedan för att demonstrera olika för- och nackdelar med båda alternativen. De serverlösa implementeringarna som presenteras i denna avhandling har utvecklats med hjälp av serverlösa plattformar som erbjuds av Microsoft Azure. De konventionella implementeringarna utnyttjar däremot Spring Boot, vilket är ett ramverk för Javaapplikationer.

## Serverlösa plattformar inom Azure

Azure är Microsofts molntjänst som lanserades kommersiellt 2010. Sedan dess har Azure blivit en av de största molntjänsterna inom branschen. De serverlösa implementeringarna till de tre användningsfallen är utvecklade med hjälp av följande serverlösa plattformar som är tillgängliga i Azure.

### Azure Functions

Functions är Microsofts FaaS-tjänst inom Azure. Tjänsten gör det möjligt för utvecklare att tillämpa affärslogik i form av funktioner, som sedan kan självständigt sättas i drift och exekveras i molnet. Liksom i andra motsvarande tjänster, ansvarar Azure för de operativa aspekterna för att exekvera koden. Functions stöder flera programmeringsspråk, bland annat C#, Java, JavaScript och Python.

### Azure Logic Apps

Logic Apps är en serverlös plattform för arbetsflöden inom Azure. Till skillnad från Azure Functions, där affärslogik tillämpas med hjälp av kod, innehåller Logic Apps i stället ett eget designverktyg för att skapa arbetsflöden. Dessa arbetsflöden byggs upp av komponenter som utvecklaren lägger till med hjälp av verktyget. Denna så kallade ”design first”-strategin gör det möjligt att tillämpa affärslogik utan att behöva skriva någon kod. Precis som i andra serverlösa plattformar hanterar molnleverantören de operativa aspekterna av att exekvera arbetsflödet.

### Azure Cosmos DB

Cosmos DB är en NoSQL-databastjänst inom Azure. Även om tjänsten inte uppfyller samma definitioner för serverlösa plattformar som beskrivits i inledningen, fungerar Cosmos DB bra tillsammans med FaaS och andra serverlösa tjänster av flera skäl. På samma sätt som i serverlösa tjänster sköter Azure de operativa aspekterna av databaserna. Dessutom är det möjligt att i tjänsten skapa databaser i ett så kallat förbrukningsbaserat läge. I detta läge behöver användaren inte i förväg estimera eller binda sig till en viss mängd kapacitet för databasen. I stället betalar användaren enbart för de resurser som förbrukats av databasoperationerna. Azure kallar därför detta läge för serverlöst.

## Användingsfallen

Användningsfallen som undersökts i denna avhandling presenteras i det följande kort sammanfattat. De tre fallen beskrivs från synvinkeln av en fiktiv exempelorganisation med namnet *The Example Organisation*, förkortningsvis bara *TEO*.

### Användningsfall 1: Integration

TEO har besultat att börja återförsälja licenser till en populär programvara. Licenserna är användarbaserade och de förvärvas samt förvaltas via en särskild marknadsplats. Varje företag som använder programvaran har en egen profil inom marknadsplatsen, där produktlicenserna finns införda för varje användare. För att TEO ska kunna fakturera sina kunder för produktlicenserna, måste en systemintegration skapas mellan marknadsplatsen och TEOs affärssystem.

I början av varje månad ska tillämpningen hämta alla licenser som använts under föregående månad från marknadsplatsen genom dess API. Den totala kostanden för licenserna ska sedan beräknas för varje kund, varefter informationen ska sparas i affärssystemet i form av en debitering. Kundens profil i marknadsplatsen och affärssystemet kopplas ihop med hjälp av företagets VAT-nummer. Ifall företagets profil inte hittas i affärssystemet med VAT-numret, ska informationen om produktlicenserna skickas till faktureringsavdelningen med e-post. Processen finns avbildad detaljerat i figur 4.1.

### Användningsfall 2: Faktureringsmotor

TEO har beslutat att börja erbjuda en egen servertjänst. Tjänsten gör det möjligt för kunder att hyra dator- och lagringsresurser av olika varianter. Båda resurstyperna faktureras utifrån reserverad kapacitet. Varje resurs har ett baspris, vilket motsvarar priset för att reservera resursen för en timme. Baspriset för varje typ av resurs anges i en prislista. Standardprislistan används i normala fall, men vissa kunder har förhandlat fram egna prislistor med andra baspriser. Alla prislistor hanteras av tjänsten *Pricing Service* och kan hämtas därifrån via dess API.

Användningsdata för de olika resurserna hanteras av en separat tjänst som kallas för *Data Collector*. Dessa användningsdata består av olika typer av händelser för resurserna, vart bland annat tillhör tidpunkterna när de skapades och raderades. Dessa så kallade *resurshändelser* lagras i tjänsten och kan hämtas med hjälp av dess API.

För att TEO ska kunna fakturera sina kunder för dessa, måste de implementera an faktureringsmotor. Motorn bör först hämta alla resurshändelser från *Data*

*Collector*, vilka används för att sedan beräkna användningstiden av resurserna. Användningstiden ska sedan multipliceras med baspriset på resursen för att bilda dess totala kostnad. Exempel på hur dessa kostnadsberäkningar sker kan ses i tabellerna 4.1 och 4.2. Dessa kostnader ska sedan sättas in i affärssystemet i form av debiteringar, liksom i förra användningsfallet. Processen finns avbildad i högre detalj i figur 4.3.

#### Användningsfall 3: API

Olika team inom TEO skapar dagligen nya kundprofiler i affärssystemet. Processen innebär att flera olika uppgifter om ett företag ska föras in i systemet, bland annat namn, adresser och telefonnummer. Denna process kan vara rätt jobbig, eftersom användaren för hand måste hämta och mata in dessa uppgifter i systemet för att skapa kundprofilen. TEO har därför identifierat detta som ett potentiellt område som kan automatiseras.

De företagsuppgifter som används för att skapa kundprofiler är allmänt tillgängliga i företagsinformationssystemet, ett offentligt register över företag som upprätthålls av en statlig myndighet. Företagsinformationssystemet har ett API som returnerar företagsuppgifter i XML-format. Affärssystemet innehåller också ett eget API, som kan användas för att skapa en ny kundprofil i systemet. API:et accepterar en JSON-fil som innehåller alla nödvändiga företagsuppgifter som behövs för att skapa en kundprofil.

För att automatisera skapandet av kundprofiler i affärssystemet måste TEO implementera ett nytt API, som senare kommer att utnyttjas av ett internt verktyg. API:et ska acceptera ett FO-nummer som parameter. Detta FO-nummer ska sedan användas för att försöka hämta information om företaget från företagsinformationssystemet. Ifall detta lyckas ska uppgifterna omvandlas från XML-formatet till JSON-formatet som accepteras av affärssystemet, varefter de ska sättas in i systemet. Processen finns avbildad i högre detalj i figur 4.5.

## Implementeringar

Inom ramen för arbetet med denna avhandling skapades en konventionell och en serverlös implementering för de tre användningsfallen. Implementeringarna beskrivs i det följande kort sammanfattat.

### Användningsfall 1

Den konventionella implementeringen följer en struktur som är allmän bland Spring Boot-applikationer. Klassdiagrammet för programmet finns avbildat i figur 5.1. Datautbyte mellan applikationen, marknadsplatsen och affärssystemet sker via deras respektive klientklasser. Dessutom innehåller implementeringen en egen klass för att skapa och skicka e-post via en särskild SMTP-server.

Den servelösa implementeringen utnyttjar huvudsakligen Azure Logic Apps. Anledningen till detta val var huvudsakligen användningsfallets relativt enkla affärslogik, vilket gjorde det möjligt att utnyttja en arbetsflödesmotor utöver traditionella programmeringsmetoder. Azure Functions utnyttjas dock också för skapa informationen för debiteringen, eftersom Logic Apps endast kan utföra enkla beräkningsoperationer. Arbetsflödet finns avbildat i flera figurer i kapitel 5.1.2.

### Användningsfall 2

Den konventionella implementeringen för detta användningsfall är på många sätt mera komplicerad jämfört med den föregående på grund av den mer invecklade affärslogiken. Utmärkande för detta användningsfall är bland annat införandet av persistenskontroll med Spring Data JPA. Ett fullständigt klassdiagram för den konventionella implementeringen är avbildat i figur 5.8. Affärslogiken är indelad i flera olika klasser, där varje klass ansvarar för ett visst område av faktureringsprocessen.

Den serverlösa implementeringen utnyttjar Azure Functions för att utföra beräkningarna. Dessutom utnyttjas även Durable Funcions, ett insticksprogram för Azure Functions som gör det möjligt att skapa invecklade arbetsflöden bestående av flera funktioner. För persistenskontroll används även Azure Cosmos DB. Själva implementeringen består av nio funktioner, vilka är avbildade på hög nivå i figur 5.9. Varje funktion har en egen roll i faktureringsprocessen. Rollen kan rätt enkelt tolkas direkt utifrån funktionens namn. En av de betydande skillnaderna jämfört med den konventionella implementeringen är införandet av parallella beräkningar, vilket möjliggjorts med hjälp av insticksprogrammet Durable Functions.

### Användningsfall 3

Det tredje användningsfallet hade den enklaste affärslogiken, vilket syns tydligt också i den konventionella implementeringens klassdiagram, avbildat i figur 5.10. Unikt för just denna implementering är användningen av bland annat Spring Security, som utnyttjas för att försäkra API:n från utomstående enheter.

Den serverlösa implementeringen använder sig enbart av en tjänst, Azure Func-

tions. Trots detta påminner den serverlösa implementeringen om den konventionella på flera sätt, eftersom båda är utvecklade med programmeringsspråket Java. Detta gjorde det möjligt att delvis återanvända kod från den konventionella implementeringen i den serverlösa, vilket gjorde själva utvecklingsprocessen kort. Ett diagram över den serverlösa implementeringen finns avbildat på hög nivå i figur 5.11.

## Fördelar

De serverlösa plattformarna har alla olika särdrag som utvecklaren måste bekanta sig med för att kunna använda dem ordentligt. Efter att detta har skett, kan plattformarnas egenskaper dock användas till nytta. Bästa exemplet på detta är parallelliseringen av beräkningarna i den serverlösa implementeringen av användningsfall 2. Där behövde utvecklaren endast skriva kod för att starta exekveringen av aktivitetsfunktionerna, medan plattformen tog hand om resten. Att utveckla liknande funktionalitet i den konventionella implementeringen är inte omöjligt, men skulle kräva betydligt större insats.

Ett annat bra exempel finns även i användningsfall 1, där implementeringarna skapar och skickar ett e-post till faktureringsavdelningen ifall undantag sker. Den konventionella implementeringen kräver att en särskild SMTP-server konfigureras i programkonfigurationerna för att skicka e-post. Den serverlösa implementeringen skickar däremot e-postmeddelande med hjälp av ett Outlook-konto, som användaren bara måste logga in med. Logic Apps tar hand om själva skickandet av e-posten i bakgrunden.

Särdragen hos de serverlösa plattformarna gör dem även väl lämpade för program med hög skuröverföringsgrad. Sådana program är i huvudsak inaktiva, men producerar mycket aktivitet när exekveringen sker. Användningsfallen 1 och 2 är båda bra exempel detta. Eftersom faktureringsmodellen av serverlösa tjänster baserar sig på bruk, betyder det att de serverlösa implementeringarna inte resulterar i några som helst operativa kostnader när de är inaktiva. Dessutom är själva exekveringskostnaderna också relativt låga. Till exempel kostar exekveringen av den serverlösa implementeringen av användningsfall 1 lite över 0,10 amerikanska dollar då tusen kunders licenser ska behandlas.

## Nackdelar

Den bruksbaserade faktureringsmodellen av serverlösa plattformar kan dock resultera i betydande kostnader ifall exekveringen sker tillräckligt ofta. Även mindre summor, som 10 cent per exekvering, kan växa till en avsevärd kostnad ifall pro-

grammet körs flera gånger i timmen. I sådana fall kan det vara förmånligare att välja reserverad kapacitet i stället.

En annan nackdel med serverlösa plattformar är att implementeringarna kan sättas i drift endast inom plattformen de är skapade för i molnet. Ofta är detta inte ett problem, men det finns fall där det kan vara nödvändigt att sätta programvara i drift på egna servrar, till exempel på grund av dataskyddskrav. De konventionella implementeringarna har inga liknande restriktioner, utan kan däremot driftsättas i praktiken i vilken miljö som helst.

De konventionella implementeringarna har också vissa andra fördelar. Även om utvecklingsprocessen av själva programmet kräver en större insats jämfört med det serverlösa alternativet, är det slutliga resultatet en fullständig Spring Boot-applikation. Detta gör det lättare att utvidga programmets funktionalitet i framtiden ifall behovet uppstår. I de serverlösa implementeringarna kan funktionaliteten också utvidgas, men det sker nödvändigtvis inte lika enkelt. Tydligast märks detta i användningsfall 1, där den serverlösa implementeringen är ett lineärt arbetsflöde i Logic Apps. Arbetsflödet går det svårare att utvidga funktionaliteten utan att behöva lägga till ett helt nytt flöde. Samma problem märks även delvis i användningsfall 3, där den serverlösa implementeringen är skapad med hjälp av Azure Functions. Även om flera funktioner kan dela klasser, måste en del funktionalitet implementeras separat för varje funktion.

Den största nackdelen med serverlösa tjänster är dock att de gör programmen stark kopplade till plattformen som de är skapade för, och därmed även molnleverantören som erbjuder tjänsten. Problemet förvärras av att de serverlösa plattformarna ofta medför ytterligare kopplingar till andra tjänster inom molnleverantörens ekosystem. Dessa kopplingar gör det svårt och dyrt att överföra programmet till en annan miljö i framtiden ifall detta av en orsak eller annan krävs. I vissa situationer, som i användningsfall 1, går det inte alls att flytta programmet, utan affärslogiken måste implementeras på nytt från början. De konventionella implementeringarna innehåller inte motsvarande kopplingar.

## Slutsats

Beslutet om huruvida man ska använda serverlösa plattformar eller inte bör därför fattas noggrant. Abstraheringen som plattformarna medför kan bidra till att minska utvecklingstiden och den operativa bördan. De serverlösa plattformarnas särdrag gör dem dessutom väl anpassade bland annat för program med hög skuröverföringsgrad. De starka kopplingarna som plattformarna medför kan däremot utgöra en affärsrisk, eftersom de gör programmen helt beroende av en viss molnleverantör.

Detta är ytterst viktigt att ta i beaktande ifall programmet är kritiskt för företagets verksamhet.

# References


[1] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță and A. Iosup, "Serverless is More: From PaaS to Present Cloud Computing", *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, Sep. 2018, Conference Name: IEEE Internet Computing, ISSN: 1941-0131.

[2] J. Nupponen and D. Taibi, "Serverless: What it Is, What to Do and What Not to Do", in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2020, pp. 49–50.

[3] I. Baldini *et al.*, "Serverless Computing: Current Trends and Open Problems", *arXiv:1706.03178 [cs]*, Jun. 2017, arXiv: 1706.03178. [Online]. Available: http://arxiv.org/abs/1706.03178 (visited on 07/02/2022).

[4] *AWS Lambda - Amazon Web Services*, en-US. [Online]. Available: https://aws.amazon.com/lambda/ (visited on 25/07/2022).

[5] *AWS Lambda – Run Code in the Cloud*, en-US, Section: Amazon DynamoDB, Nov. 2014. [Online]. Available: https://aws.amazon.com/blogs/aws/run-code-cloud/ (visited on 25/07/2022).

[6] *Azure Functions – Serverless Functions in Computing | Microsoft Azure*, en. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/ (visited on 25/07/2022).

[7] *Cloud Functions*, en. [Online]. Available: https://cloud.google.com/functions (visited on 25/07/2022).

[8] *Logic App Service – IPaaS | Microsoft Azure*, en. [Online]. Available: https://azure.microsoft.com/en-us/services/logic-apps/ (visited on 25/07/2022).

[9] A. Kumar and S. Mahendrakar, *Serverless Integration Design Patterns with Azure: Build Powerful Cloud Solutions That Sustain Next-Generation Products*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2019, ISBN: 978-1-78839-083-5. (visited on 09/02/2022).

[10] *AWS Lambda Pricing – Amazon Web Services*, en-US. [Online]. Available: https://aws.amazon.com/lambda/pricing/ (visited on 25/07/2022).

[11] *Azure Functions scale and hosting*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale (visited on 25/07/2022).

[12] *Lambda quotas - AWS Lambda*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html (visited on 25/07/2022).

[13] V. Yussupov, U. Breitenbücher, F. Leymann and C. Müller, "Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends", in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19, New York, NY, USA: Association for Computing Machinery, Dec. 2019, pp. 273–283, ISBN: 978-1-4503-6894-0. [Online]. Available: https://doi.org/10.1145/3344341.3368813 (visited on 17/02/2022).

[14] J. Opara-Martins, R. Sahandi and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing", in *International Conference on Information Society (i-Society 2014)*, Nov. 2014, pp. 92–97.

[15] V. Yussupov, U. Breitenbücher, F. Leymann and M. Wurster, "A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools", en, in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, Auckland New Zealand: ACM, Dec. 2019, pp. 229–240, ISBN: 978-1-4503-6894-0. [Online]. Available: https://dl.acm.org/doi/10.1145/3344341.3368803 (visited on 16/02/2022).

[16] G. C. Fox, V. Ishakian, V. Muthusamy and A. Slominski, "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research", *arXiv:1708.08028 [cs]*, 2017, arXiv: 1708.08028. [Online]. Available: http://arxiv.org/abs/1708.08028 (visited on 16/02/2022).

[17] E. Marin, D. Perino and R. Di Pietro, *Serverless Computing: A Security Perspective*, arXiv:2107.03832 [cs], Jan. 2022. [Online]. Available: http://arxiv.org/abs/2107.03832 (visited on 15/07/2022).

[18] W. O'Meara and R. G. Lennon, "Serverless Computing Security: Protecting Application Logic", in *2020 31st Irish Signals and Systems Conference (ISSC)*, ISSN: 2688-1454, Jun. 2020, pp. 1–5.

[19] S. Eismann *et al.*, "The State of Serverless Applications: Collection, Characterization, and Community Consensus", *IEEE Transactions on Software Engineering*, pp. 1–1, 2021, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520.

[20] J. M. Hellerstein *et al.*, *Serverless Computing: One Step Forward, Two Steps Back*, arXiv:1812.03651 [cs], Dec. 2018. [Online]. Available: http://arxiv.org/abs/1812.03651 (visited on 15/07/2022).

[21] JetBrains, *Programming languages used by software developers worldwide as of 2021, by deployment type*, en, Jul. 2021. [Online]. Available: https://www.statista.com/statistics/869092/worldwide-software-developer-survey-languages-used/ (visited on 27/07/2022).

[22] C. F. Foundation, "These Are the Top Languages for Enterprise Application Development - And What That Means for Business", Tech. Rep., 2018, p. 8. [Online]. Available: https://www.cloudfoundry.org/wp-content/uploads/Developer-Language-Report_FINAL.pdf (visited on 27/07/2022).

[23] B. Vermeer, "JVM Ecosystem Report 2020", Snyk, Tech. Rep., 2020. [Online]. Available: https://snyk.io/wp-content/uploads/jvm_2020.pdf (visited on 27/07/2022).

[24] *What is Java Spring Boot?—Intro to Spring Boot | Microsoft Azure*, en. [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-spring-boot/ (visited on 23/06/2022).

[25] *Spring.io*, en. [Online]. Available: https://spring.io/ (visited on 26/07/2022).

[26] C. Mehta, S. Shah, P. Shah, P. Goswami and D. Radadiya, *Hands-On High Performance with Spring 5: Techniques for Scaling and Optimizing Spring and Spring Boot Applications*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2018, ISBN: 978-1-78883-063-8. (visited on 23/06/2022).

[27] D. Rajput, *Mastering Spring Boot 2. 0: Build Modern, Cloud-Native, and Distributed Systems Using Spring Boot*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2018, ISBN: 978-1-78712-514-8. (visited on 23/06/2022).

[28] P. Webb and D. Syer, *Spring Boot – Simplifying Spring for Everyone*, en, Aug. 2013. [Online]. Available: https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone/ (visited on 27/07/2022).

[29] D. Hauger, *Windows Azure General Availability*, en-US, Feb. 2010. [Online]. Available: https://blogs.microsoft.com/blog/2010/02/01/windows-azure-general-availability/ (visited on 11/02/2022).

[30] F. Richter, *Infographic: Amazon Leads $180-Billion Cloud Market*, en. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (visited on 10/02/2022).

[31] M. Morar, A. Kumar, M. Abbott, G. K. Gautam, J. Corbould and A. Bhambhani, *Robust Cloud Integration with Azure*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2017, ISBN: 978-1-78646-818-5. (visited on 09/02/2022).

[32] *Triggers and bindings in Azure Functions*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings (visited on 28/07/2022).

[33] *Supported languages in Azure Functions*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages (visited on 28/07/2022).

[34] *Manage your function app*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-how-to-use-azure-function-app-settings (visited on 28/07/2022).

[35] D. Bass, *Advanced Serverless Architectures with Microsoft Azure: Design Complex Serverless Systems Quickly with the Scalability and Benefits of Azure*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2019, ISBN: 978-1-78839-557-1. (visited on 10/02/2022).

[36] *Durable Functions types and features*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-types-features-overview (visited on 28/07/2022).

[37] *Azure Integration Services Whitepaper*, en, Nov. 2018. [Online]. Available: https://azure.microsoft.com/en-au/resources/azure-integration-services/en-us/ (visited on 27/01/2022).

[38] *Single-tenant versus multi-tenant and integration service environment for Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/logic-apps/single-tenant-overview-compare (visited on 27/07/2022).

[39] *About connectors in Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/connectors/apis-list (visited on 27/07/2022).

[40] *Built-in connectors in Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/connectors/built-in (visited on 28/07/2022).

[41] *Managed connectors in Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/connectors/managed (visited on 28/07/2022).

[42] *Connect to Outlook.com from Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-outlook (visited on 28/07/2022).

[43] *Connect to Slack from Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-slack (visited on 28/07/2022).

[44] *Usage metering, billing, and pricing models for Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-pricing (visited on 28/07/2022).

[45] *Integration and automation platform options in Azure*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-compare-logic-apps-ms-flow-webjobs (visited on 27/07/2022).

[46] *Power Automate vs Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/microsoft-365/community/power-automate-vs-logic-apps (visited on 27/07/2022).

[47] M. J. Foley, *Where did Microsoft's new Flow event-automation service come from?*, en, May 2016. [Online]. Available: https://www.zdnet.com/article/where-did-microsofts-new-flow-event-automation-service-come-from/ (visited on 25/01/2022).

[48] *Share a cloud flow*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/power-automate/create-team-flows (visited on 28/07/2022).

[49] *Manage connections in Power Automate*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/power-automate/add-manage-connections (visited on 28/07/2022).

[50] *Types of Power Automate licenses*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/power-platform/admin/power-automate-licensing/types (visited on 27/07/2022).

[51] *Grant users access*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/power-platform/admin/grant-users-access (visited on 27/07/2022).

[52] *Introduction to Azure Cosmos DB*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction (visited on 28/07/2022).

[53] *Consumption-based serverless offer in Azure Cosmos DB*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/serverless (visited on 28/07/2022).

[54] G. C. Hillar and D. Yöndem, *Guide to NoSQL with Azure Cosmos DB: Work with the Massively Scalable Azure Database Service with JSON, C#, LINQ, and . NET Core 2*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2018, ISBN: 978-1-78961-896-9. (visited on 28/07/2022).

[55] *Choose an API in Azure Cosmos DB*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/choose-api (visited on 23/08/2022).

[56] D. Shukla, *A technical overview of Azure Cosmos DB*, en, May 2017. [Online]. Available: https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/ (visited on 23/08/2022).

[57] *Request Units in Azure Cosmos DB*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/request-units (visited on 23/08/2022).

[58] *Pricing - Azure Cosmos DB | Microsoft Azure*, en. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/cosmos-db/ (visited on 11/09/2022).

[59] *HTTP APIs in Durable Functions*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-http-api (visited on 02/08/2022).

[60] *Run actions based on group status by using scopes in Azure Logic Apps*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-control-flow-run-steps-group-scopes (visited on 30/05/2022).

[61] *SMTP - Connectors*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/connectors/smtp/ (visited on 30/05/2022).

[62] *Pricing - Logic Apps | Microsoft Azure*, en. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/logic-apps/ (visited on 19/08/2022).

[63] *Office 365 Outlook - Connectors*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/connectors/office365connector/ (visited on 19/08/2022).

[64] *Azure Cosmos DB input binding for Functions 2.x and higher*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-cosmosdb-v2-input (visited on 19/08/2022).

[65] *Pricing - Functions | Microsoft Azure*, en. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/ (visited on 15/09/2022).

[66] *How to choose between provisioned throughput and serverless on Azure Cosmos DB*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/throughput-serverless (visited on 28/07/2022).

[67] D. Everson, L. Cheng and Z. Zhang, "Log4shell: Redefining the Web Attack Surface", en, p. 8,

[68] K. Ren, C. Wang and Q. Wang, "Security Challenges for the Public Cloud", *IEEE Internet Computing*, vol. 16, no. 1, pp. 69–73, Jan. 2012, Conference Name: IEEE Internet Computing, ISSN: 1941-0131.

[69] H. H. Abraha, "How compatible is the US 'CLOUD Act' with cloud computing? A brief analysis", en, *International Data Privacy Law*, vol. 9, no. 3, pp. 207–215, Aug. 2019, ISSN: 2044-3994, 2044-4001. [Online]. Available: https://academic.oup.com/idpl/article/9/3/207/5532213 (visited on 31/08/2022).